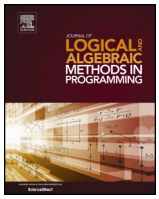
Journal of Logical and Algebraic Methods in Programming 113 (2020) 100543

Contents lists available at ScienceDirect

Journal of Logical and Algebraic Methods in Programming

www.elsevier.com/locate/jlamp



Specifying with syntactic theory functors

Magne Haveraaen [a], Markus Roggenbach [b]

[a] Bergen Language Design Laboratory, Department of Computer Science, University of Bergen, Norway
[b] Department of Computer Science, Swansea University, Wales, UK

ARTICLE INFO

Article history:
Received 9 April 2019
Received in revised form 25 March 2020
Accepted 27 March 2020
Available online 2 April 2020

Keywords:
Specification languages
Reuse mechanisms
Institution-independence

ABSTRACT

We propose a framework, *syntactic theory functors* (STFs), for creating syntactic structuring mechanisms for specification languages. Good support for common reuse patterns is important for systematically developing specifications for large systems. Though immaterial to foundational theory, lack of support otherwise causes lengthy writing of boilerplate code or repeated adaptation from one context to another.

We present STFs in the context of the Goguen & Burstall institution theory. This theory captures the essential structure of ontologies, modelling and formal specifications (OMS). In particular it provides powerful structuring mechanisms that are independent of the specification formalism, i.e., they are institution-independent.

The presented STF framework is *institution-independent* as well. As such it encompasses many approaches to software and information systems. STFs subsume the standard institution-independent structuring mechanisms, and open up new ways of reusing existing and structuring new specifications. In this, STFs subsume and enrich the tool-set of 'good practices', which includes separation of concerns, ease of reuse of specification-text, and improved theorem proving support. STFs are aimed at structuring and reuse beyond the classical mechanisms.

However, most STFs are institution-specific and support specific reuse patterns in that institution. With such institution-specific STFs it is possible to incrementally grow more complex institutions from simpler ones. This is very much needed when developing ontologies or specification languages for a new domain.

In this paper, we motivate STFs with examples in CASL, the common standard algebraic specification language. We further demonstrate how STFs can ease specification through capturing repeated constructions once and for all as patterns formulated as STFs.

1. Introduction

Developing formal specifications for practical problems hits issues that are immaterial to foundational specification theory. One issue is that it might be awkward to specify a simple idea, e.g., writing axioms in first-order-logic expressing that a remote control must have at least 13 distinct buttons. Another issue is that several specifications are conceptually the same, with minor differences in declarations causing them to be duplicated. One might, for instance, want to specify a device similar to an already specified one, but with two more buttons. Or one might want to expand a data type with partial functions and handle partiality with error elements, which forces all existing operations also to deal with the error

E-mail address: magne.haveraaen@ii.uib.no (M. Haveraaen).
URLs: https://bldl.ii.uib.no/ (M. Haveraaen), http://www.cs.swan.ac.uk/~csmarkus/ (M. Roggenbach).

https://doi.org/10.1016/j.jlamp.2020.100543

elements. Or adding an extra argument to a function, but keep the axioms. The list of such examples is endless. However, the current lack of language support to deal with situations like this causes lengthy writing of boilerplate code or repeated adaptation from one context to another.

The theme common to these examples is that they are of transformational nature. Given a specification, one would like to transform it into a different one, e.g., the empty specification to one where there are 13 distinct buttons, a data type specification to one involving an error element. Here, different contexts might ask for the application of essentially the 'same' transformations. Transformations that are conceptually simple, but often prone to errors to get right.

Thus, we propose that specifications involving such patterns should be automatically generated. More concisely, we claim that, like in programming, also in specification there are recurring problems for which there exist 'standard solutions'. On the conceptual side, we call these solutions 'patterns'. On the technical side, we realise such 'patterns' by transformations. Concretely, we suggest the notion of *syntactic theory functors* (STFs) to support such transformations.

We study and develop syntactic theory functors in the context of Goguen & Burstall's institution theory, and illustrate them with examples developed in the algebraic specification language CASL.

1.1. Design patterns and how to formalise them

Design patterns have been a longstanding topic, both in programming and in formal specification. Their purpose is to capture standard solutions for recurring challenges that a programmer or specifier might face. However, while design patterns in programming, cf. [1], are already an established topic which regularly is taught in software engineering courses, how to deal with specification patterns is still an emerging research topic.

In the context of the algebraic specification language CASL [2], Cerioli and Reggio study the interplay between partiality, subsorting and predicates [3]. The purpose of their investigation is to systematically guide the specifier with the help of decision diagrams to the 'best' solution. However, their focus is on guidance rather than on how to formulate patterns.

Also in the context of CASL, "Methodological guidelines" [4] accompany the library of CASL standard datatypes, Part V in [2]. These guidelines are written in controlled language adhering to a strict format, which was inspired by the presentation style of "A Pattern Language" [5]. The purpose of these guidelines was to apply uniform solutions to similar challenges throughout a comprehensive library. Their focus is on identifying specification challenges and arguing why the specific chosen solution is a good one.

In the context of ontology design, various patterns have been described, identified, named, and established. There is an "Association for Ontology Design & Patterns (ODPA)", which focuses on methods and tools arising out of ontology design patterns.[1] ODPA runs the annual workshop on ontology design and patterns (WOP), in 2019 in its 10th edition. Examples of ontology design patterns are the role pattern or the time pattern, cf. [6]. The general objective is to "enable domain ontologists to reuse existing best practices and design decisions and, thus, benefit from the experience of ontology experts" [6]. This overall goal is addressed with different techniques. Krieg-Brückner et al. use institution-independent structuring mechanisms, in particular generic specification, in order to obtain transformations realising the patterns. We considered this approach as well, however, dismissed it as not flexible enough – see Section 2.1.

1.2. Requirements on specification transformations

One natural requirement on specification transformations is that they shall be 'flattenable,' i.e., be syntactic sugar only. On the conceptual side this aligns with the observation that the issues to be addressed are immaterial to foundational specification theory. In specification practice, flattenable transformations allow the specifier to expand them like a 'macro,' possibly with tool support. This allows the specifier to check if the generated axioms actually look as expected.

Yet another desirable element of specification transformations is that they shall take, e.g., selected signature elements (such as the sort and a list of constants for defining any number of distinct constants) as their arguments. Such arguments would 'steer' the transformations' use in the concrete application context. For example, we can lift a set of functions on elements to functions on arrays. An n-ary function would lift to a function on n arrays. To specify the effect of lifting, the lifting transformation needs to identify the element type and set of functions to be lifted, as well as the array type, the index type and the indexing function. Thus, we want to describe families of specification transformations rather than single transformations.

In this paper, we propose a framework for creating institution-specific structuring mechanisms, *syntactic theory functors* (STFs). STFs subsume the standard institution-independent structuring mechanisms, and open up new ways of reusing existing and structuring new specifications. This allows building a rich institution incrementally from a simple institution and appropriate STFs, very much needed when developing ontologies or specification languages for a new domain. Richer institutions yield more versatility to the Distributed Ontology, modelling and specification Language (DOL) [7], which provides structuring and interoperability between any institution-based formalism.

[1] http://ontologydesignpatterns.org/wiki/ODPA.

1.3. Overview

This paper is organised as follows. First, we present a number of motivating examples exhibiting a number of 'specification patterns'. Then, in Section 3, we provide our suggestion of how to capture 'specification patterns', namely by defining *syntactic theory functors* (STFs). We establish basic properties of STFs, relate them to the similar concept of theoroidal (co)morphisms, show that established structuring mechanisms are STFs, and, finally, that our examples can be treated with STFs. In Section 4, we establish that STFs can be considered as a new language to structure specifications. We apply STFs to a case study and discuss methodological implications. We then study STFs on a more fundamental level in Section 5. We establish in what sense the STF examples of this paper reflect and preserve properties. Furthermore, we show that the so-called model consistent STFs give rise to an institution. The following section looks at the pragmatics of using STFs. In Section 7, we review the literature concerning other potential solutions, followed by the final summary section.

2. Motivating examples

Here we present some examples of recurring challenges, which, for ease of understanding, we formulate in the first order sublanguage of the algebraic specification language CASL [2]. This sublanguage is significant as, e.g., it is supported by automated theorem provers.

2.1. Finite set of distinct constants

Specifying that a simple remote control has at least a given set of distinct buttons turns out to be awkward in the first order sublanguage of CASL. The issue being that every button has to be stated as distinct from every other button, giving growth in *distinctness axioms* proportional to the square of the number of constants.

spec SORTWITHMINIMUM5ELEMENTS =
  sort s
  ops $c1, c2, c3, c4, c5 : s$
  • $\neg\ c1 = c2 \wedge \neg\ c1 = c3 \wedge \neg\ c1 = c4 \wedge \neg\ c1 = c5 \wedge$
    $\neg\ c2 = c3 \wedge \neg\ c2 = c4 \wedge \neg\ c2 = c5 \wedge$
    $\neg\ c3 = c4 \wedge \neg\ c3 = c5 \wedge$
    $\neg\ c4 = c5$
end

Writing this standard pattern of axioms is awkward when the number of constants grows, and may easily introduce errors in a conceptually simple specification. We propose that such patterns can be automatically generated by an STF *free* for a given set of constant declarations.[2]

The STF *free* constructs a specification depending on a given set of constant symbols; i.e., *free* actually represents a *family* of constructions, where the indices are sets of qualified symbol names. Such STFs can be combined as the need for constants grows, e.g., starting with a remote control for volume control (distinct buttons for volume up and down), but later extended to a more complex remote control with more buttons for more functionality.

2.1.1. Dismissal of the CASL parameterization mechanism

Alternatively to an STF *free*, one might be tempted to utilize CASL parameterization and instantiation by providing a library that consists of family of specifications as follows:

spec SORTWITHMINIMUM2ELEMENTS [**sort** s **ops** $c1, c2 : s$] =
  • $\neg\ c1 = c2$
end

spec SORTWITHMINIMUM3ELEMENTS [**sort** s **ops** $c1, c2, c3 : s$] =
  • $\neg\ c1 = c2 \wedge \neg\ c1 = c3 \wedge$
    $\neg\ c2 = c3$
end

...

Such parametrised CASL specifications can be instantiated, e.g., in order to specify that for the purpose of volume control the remote shall have two buttons "up" and "dn", which are different from each other:

[2] Note that the CASL tool Hets [8] supports *free types* (freely generated types) which can define a finite number of constants, while our *free* construction defines a *minimal* number of constants. The CASL 'free type' can be seen as a combination of an STF *free* and an STF *generated*.

spec VolumeControl = SortWithMinimum2Elements
[**sort** *VolumeButtons* **ops** *up*, *dn* : *VolumeButtons*
fit ops *c1* ↦ *up*, *c2* ↦ *dn*]
end

On these specifications we can observe:

- CASL parameterization *fixes* the number of symbols in the parameter. There are either two, three, … constants that are declared in the formal parameter. These can be reused in any context using a signature morphism, with the induced contravariant model functor transporting models consistently across from the larger context to the component specification.
- There is no language support in CASL to express that the specifications SortWithMinimum2Elements, SortWithMinimum3Elements, … have anything to do with each other: they could stand for completely unrelated concepts.

The first point limits expressivity. As a library can only comprise of finitely many specifications, in specification practice some size is bound to be missing. The second point makes it hard to prove theorems on such a family of specifications as the underlying 'construction principle' of the specification sequence has not been expressed. For instance, we can show that free in n constants implies free in m constants for $m \leq n$ (assuming the set of m constants is included among the n constants).[3] In Section 5.2, we will prove 'construction principle' theorems for families of STFs.

Overall, we conclude that CASL parametrization fails to be well-suited for expressing 'specification patterns'.

2.2. Generatedness

Related to the free STF, we should also consider an STF *generated* for limiting a type's carrier set to that of the available constants.

spec SortWithMaximum5Elements =
sort s
ops $c1, c2, c3, c4, c5 : s$
• $\forall\, x : s \bullet x = c1 \vee x = c2 \vee x = c3 \vee x = c4 \vee x = c5$
end

Here we need one formula with as many disjunctions as the number of elements we are limiting the carrier to have. Similar to the meta-theory of the *free* construction, we can show that generated in m constants implies generated in n constants for $m \leq n$ (assuming the set of m constants is included among the n constants).

Combining the free and generated functors (in either order) for the same set of constants, gives a specification of a type with exactly the given number of elements. If the generated (maximum) number is higher than the free (minimal) number of constants, there are models with varied sets of elements. On the other hand, if the generated number of elements is strictly lower than the free number of elements, the combination is inconsistent and has no models. This shows that an STF, just like the standard institution-independent union operation on specifications, changes the model class in 'arbitrary' ways.

2.3. Adding arguments to declarations

Another recurring issue is when we basically want the same function declarations and the same axioms as in another specification, the only difference being that the functions have extra arguments. This augmenting of argument lists with extra parameters controls the 'specialisation' or 'direction' of the functions.

We illustrate this by an example from differential algebra. The Leibniz rule is taken as defining a derivative D on a ring r, here formulated also with a directional parameter *dir* to define partial derivatives [9].

spec LeibnizRule =
sort r
ops $\_+\_, \_*\_ : r \times r \to r$;
$D : r \to r$
$\forall\, x, y : r$
• $D(x + y) = D(x) + D(y)$
• $D(x * y) = D(x) * y + x * D(y)$
end

spec DirectionalLeibnizRule =
sorts *dir*, r
ops $\_+\_, \_*\_ : r \times r \to r$;
$D : dir \times r \to r$
$\forall\, d : dir; x, y : r$
• $D(d, x + y) = D(d, x) + D(d, y)$
• $D(d, x * y) = D(d, x) * y + x * D(d, y)$
end

[3] Akin to the benefits of the meta-theory of CASL 'free types' as opposed to always working on specific instances of the theory.

The augmented specification clearly allows for a richer class of models than the original specification. Only if the D function on the right was forced to yield the same value for all directions, i.e., $D(a, x) = D(b, x)$, would there be an isomorphism between the two model classes. However, for any fixed direction d, there is a projection from DirectionalLeibnizRule into LeibnizRule given by $D_d : r \to r$ (replacing D on the left) defined by $D_d(x) = D(d, x)$ (from the right). This kind of play with notation, moving an argument down as an index on a function or moving an index up as an argument, is common in mathematics. In computing, moving between notation is not that simple, and a mechanism for syntactically modifying specifications in this way is needed.[4] Here an *augment* STF should be able to generate DirectionalLeibnizRule from LeibnizRule avoiding future maintenance problems.

Note that the corresponding construction will augment a selected subset of functions, cf. Section 3.4 below.

2.4. Lifting operators

Functional programmers have demonstrated the convenience of mapping functions across lists, e.g., having a function that doubles integers be mapped across a list to double all element values of that list. In *array-oriented programming* mapping n-ary functions across n arrays is a basic operation. Depending on properties of the elemental functions, the mapped functions will inherit similar properties. For arrays we need an index type I and an index function *get* that gets the element E from an array A.

spec BinaryMap =
  sorts A, E, I
  op $f : A \times A \to A$
  op $f : E \times E \to E$
  op $get : A \times I \to E$
  $\forall\ a, b : A; i : I \bullet get(f(a, b), i) = f(get(a, i), get(b, i))$ % lifting
  $\forall\ a, b : A \bullet a = b \Leftrightarrow \forall\ i : I \bullet get(a, i) = get(b, i)$ % partitioning
end

This specification states that mapping a binary function f across two arrays is applying the elemental function to each corresponding element of the two argument arrays. Furthermore, a partitioning axiom states that equality on arrays is defined component-wise.

Note that the corresponding construction can lift any function, cf. Section 3.4 below, though our motivating example demonstrates it for one binary function only. Given axioms on E, some of these will be valid for the array as well.

2.4.1. Properties can be preserved

Equational formulae on the elemental functions are valid for the mapped functions. For instance, commutativity holds for the lifted operation when it holds for the elements.

spec CommutativeMap =
  BinaryMap
then op $f : E \times E \to E, comm$
then %implies
  op $f : A \times A \to A, comm$
end

The corresponding STF *lift* can take a specification, e.g., that the elements form a ring, the declaration of an indexing function $get : A \times I \to E$, and generate the partitioning axiom, the mapping of all the elemental operations, and all implied equational properties of the mapped functions, i.e., in our case that the arrays also form a ring.

2.4.2. Properties can be destroyed

The integers are an integral domain, i.e., the following property holds:

$$\forall\ x, y : int \bullet x * y = 0 \Rightarrow x = 0 \vee y = 0$$

However, when applying lifting to the integers, the lifted variant of this property does not hold. Take the model of BinaryMap where sort E is interpreted by the integers, and the arrays A are integer vectors with two components. Then we obtain through lifted multiplication

$$(0, 1) * (1, 0) = (0, 0)$$

though both of the factors are different from $(0, 0)$.

[4] For software development, modern integrated development environments (IDEs) such as Eclipse have support for such refactoring.

2.5. Partiality

Introducing partiality into total algebraic specifications is an error-prone activity. Mosses [10] presents a few such approaches with a discussion of some of the pitfalls. Here we use the idea of an error algebra in order to show the technicalities involved in changing a simple specification to one that contains error elements.

Let us start with the specification of a commutative ring,

$$\begin{aligned}
(a+b)+c &= a+(b+c),\\
a+b &= b+a,\\
a+0 &= a,\\
a+(-a) &= 0,\\
(a*b)*c &= a*(b*c),\\
1*a &= a,\\
a*b &= b*a,\\
a*(b+c) &= a*b+a*c.
\end{aligned}$$

From this we can easily prove that $a*0=0$ by the following sequence $0 = a*0+(-(a*0)) = a*(0+0)+(-(a*0)) = a*0+a*0+(-(a*0)) = a*0$.

Now we can turn the commutative ring into a field by adding an inverse function and the conditional equation $a \neq 0 \Rightarrow a*\mathrm{inv}(a)=1$. This leaves open of what happens to $\mathrm{inv}(0)$. Given a total ring algebra we can turn it into a partial algebra, by keeping the carrier set and all functions from the total algebra, add the inverse as given by the conditional axiom, and just leave $\mathrm{inv}(0)$ undefined.

If we want to stay within total algebras, one way to handle $\mathrm{inv}(0)$ is to introduce an error element and state that the inverse of $\mathrm{inv}(0) = \mathrm{error}$. Wanting to propagate the error element through the other operations, we can add axioms,

$$\begin{aligned}
a+\mathrm{error} &= \mathrm{error},\\
-\mathrm{error} &= \mathrm{error},\\
\mathrm{error}*a &= \mathrm{error},\\
\mathrm{inv}(\mathrm{error}) &= \mathrm{error}.
\end{aligned}$$

Unfortunately this has the consequence that $a=0$ for every element: $0 = \mathrm{error}*0 = \mathrm{error}$ and $a = a+0 = a+\mathrm{error} = \mathrm{error}$. The problem is that the ring axioms imply that 0 is an absorbing element for multiplication (only) and the new axioms define error as an absorbing element for all operations. Since absorbing elements are unique, this forces $0 = \mathrm{error}$, which then has the consequence that 0 becomes absorbing and neutral for addition, which is only possible in the trivial ring.

To avoid this we must guard the original axioms with the condition that they only work when the arguments are not the error element, and then add a complete set of error propagation rules,

$$\begin{aligned}
a\neq\mathrm{error}\wedge b\neq\mathrm{error}\wedge c\neq\mathrm{error} &\Rightarrow (a+b)+c = a+(b+c),\\
a\neq\mathrm{error}\wedge b\neq\mathrm{error} &\Rightarrow a+b = b+a,\\
a\neq\mathrm{error} &\Rightarrow a+0 = a,\\
a\neq\mathrm{error} &\Rightarrow a+(-a) = 0,\\
a\neq\mathrm{error}\wedge b\neq\mathrm{error}\wedge c\neq\mathrm{error} &\Rightarrow (a*b)*c = a*(b*c),\\
a\neq\mathrm{error} &\Rightarrow 1*a = a,\\
a\neq\mathrm{error}\wedge b\neq\mathrm{error} &\Rightarrow a*b = b*a,\\
a\neq\mathrm{error}\wedge b\neq\mathrm{error}\wedge c\neq\mathrm{error} &\Rightarrow a*(b+c) = a*b+a*c,\\
a\neq\mathrm{error}\wedge a\neq 0 &\Rightarrow a*\mathrm{inv}(a) = 1,\\
\mathrm{inv}(0) &= \mathrm{error},\\
a+\mathrm{error} &= \mathrm{error},\\
\mathrm{error}+a &= \mathrm{error},\\
-\mathrm{error} &= \mathrm{error},
\end{aligned}$$

$$a * \text{error} = \text{error},$$
$$\text{error} * a = \text{error},$$
$$\text{inv}(\text{error}) = \text{error}.$$

For this special case the axiom list can be simplified since associativity, commutativity, neutrality and distributivity hold also if one or more of the arguments is the error element. The inverse rules for $+$ and $*$ do not hold if a is the error element.

An *error* STF could add an explicit error element and transform formulae as demonstrated.

Adding an error element is doing a bottom completion of the partial algebra where the expression inv(0) is undefined. Bottom completion defines an adjoint situation between the partial and the total algebra, and does not in general preserve the axioms from the partial algebra to the bottom completed total algebra. Conditionals avoiding error arguments in the axioms are needed.

3. Syntactic theory functors

The examples in the previous section show that there is a varied need for transporting specifications from the base context to a new context. This breaks the general requirement for institution-independent structuring mechanisms which have to be compatible with the signature morphisms for all signatures.

In this section, we give a definition of STFs in the context of institution theory and prove some elementary properties of STFs. This is followed by a discussion of an alternative approach to reuse. In Subsection 3.3 we prove that the kernel language of syntactic structuring mechanisms, the so-called institution-independent structuring mechanisms, are STFs. We then show that our motivating examples fit into our definition. In the following section, Section 4, we demonstrate that STFs subsume a general class of specification building operations that have been studied in the literature, namely the so-called syntactic structure mechanism.

3.1. Definition of STFs and their elementary properties

Here we discuss how to capture construction principles of specifications in an institution-independent way. We are looking for a flexible framework that is capable of capturing the many transformations of specification text that the specifier might want to carry out. We first recall the definition of an institution:

Definition 1. An institution $\mathcal{INST} = \langle \mathbf{Sig}, \text{for} : \mathbf{Sig} \to \mathbf{Set}, \text{mod} : \mathbf{Sig}^{\text{op}} \to \mathbf{CAT}, \models \rangle$ consists of

- a category **Sig** of signatures,
- a functor for : **Sig** $\to$ **Set** of formulae,
- a contravariant functor mod : **Sig**$^{\text{op}}$ $\to$ **CAT** of models,
- a satisfaction relation $\models$ which for every object Σ of **Sig** defines a relation $\models_\Sigma \subseteq \text{mod}(\Sigma) \times \text{for}(\Sigma)$,

such that the satisfaction condition holds: for every morphism $\sigma : \Sigma \to \Sigma'$ in **Sig**,

$$\left(\text{mod}(\sigma^{\text{op}} : \Sigma \leftarrow \Sigma')\right)(M') \models_\Sigma \varphi \iff M' \models_{\Sigma'} \left(\text{for}(\sigma : \Sigma \to \Sigma')\right)(\varphi) \tag{1}$$

where M' is an object of $\text{mod}(\Sigma')$, and $\varphi \in \text{for}(\Sigma)$.

A formula for a given signature is a *tautology* if it always holds. A set of formulae is *non-trivial* if at least one formula is not a tautology.

Objects of our discourse will be theories, more concisely the objects of a discrete theory category.[5] As we shall see in Section 3.3, this is in line with the institution-independent structuring mechanisms.

Definition 2. The *discrete theory category* $\mathbf{Th}_{\text{id}}$ for an institution $\mathcal{INST}$ has theories $\langle \Sigma, \Phi \rangle$, where Σ is a signature and $\Phi \subseteq \text{for}(\Sigma)$ is a set of formulae for that signature, as objects, and the identity morphisms $\text{id} : \langle \Sigma, \Phi \rangle \to \langle \Sigma, \Phi \rangle$ as morphisms.

The *discrete signature category* $\mathbf{Sig}_{\text{id}}$ for an institution $\mathcal{INST}$ has signatures Σ as objects and as morphisms the identity morphisms $\text{id} : \Sigma \to \Sigma$.

We can decompose a tuple, such as a theory, into its component parts using projection functions $\_1$ to access the first component, $\_2$ to access the second component, etc. So for a theory $T = \langle \Sigma, \Phi \rangle$, the first projection $T_1 = \Sigma$ and the second projection $T_2 = \Phi$.

[5] A discrete category is a category whose only morphisms are the identity morphisms. Category theory characterizes objects in terms of the relationships, i.e., morphisms, that each object exhibits to other objects in the universe of discourse. In this sense a discrete category is a category where objects fail to have a 'social life' – to use a metaphor coined by the logician Jean-Yves Girard, which was later picked up by the computer scientist José Luiz Fiadeiro in his book "Categories for Software Engineering" [11].

The model functor is trivially extended to the discrete theory category, $\mathrm{mod} : \mathbf{Th}_{\mathrm{id}}^{\mathrm{op}} \to \mathbf{CAT}$. It is the full subcategory of models for each Σ such that each model satisfies the axioms Φ, with the identity functor on these subcategories, i.e., for each theory $\langle \Sigma, \Phi \rangle$,

$$\mathrm{mod}(\Sigma, \Phi) = \{M \in \mathrm{mod}(\Sigma) \mid M \models_\Sigma \Phi\} \subseteq \mathrm{mod}(\Sigma),$$
$$\mathrm{mod}(\mathrm{id}^{\mathrm{op}} : \langle \Sigma, \Phi \rangle \leftarrow \langle \Sigma, \Phi \rangle) = id_{\mathrm{mod}(\Sigma)} : \mathrm{mod}(\Sigma, \Phi) \to \mathrm{mod}(\Sigma, \Phi).$$

We will define the STFs to be functors on the discrete theory category. Later we will show how STFs can be used to restore the signature morphisms and the social life of theories.

Example 1. A discrete theory functor, i.e., a functor $T : \mathbf{Th}_{\mathrm{id}} \to \mathbf{Th}_{\mathrm{id}}$, can map any theory to any theory. Take for instance the family of functors T described by

$$T(\Sigma, \Phi) = \begin{cases} \langle \Sigma', \Phi' \rangle & \text{when } \Phi = \emptyset \text{ for some signature } \Sigma' \neq \Sigma \text{ and some } \Phi' \\ \langle \Sigma'', \emptyset \rangle & \text{when } \Phi \neq \emptyset \text{ and } \Phi \text{ consistent for some signature } \Sigma'' \neq \Sigma \\ \langle \Sigma, \Phi \rangle & \text{when } \Phi \neq \emptyset \text{ and } \Phi \text{ inconsistent} \end{cases}$$

Such 'unregulated' functors have few interesting properties. Therefore, we suggest a restricted version of the discrete theory functor.

Definition 3 *(Syntactic theory functor (STF)).* We call a discrete theory functor $F : \mathbf{Th}_{\mathrm{id}} \to \mathbf{Th}_{\mathrm{id}}$ *syntactic* if the application of F on objects can be decomposed

$$F(\Sigma, \Phi) = \langle F_{\mathtt{sig}}(\Sigma), F_{\mathtt{base}}(\Sigma) \cup F_{\mathtt{for},\Sigma}(\Phi) \rangle$$

where

- $F_{\mathtt{sig}} : \mathbf{Sig}_{\mathrm{id}} \to \mathbf{Sig}_{\mathrm{id}}$ is a functor,
- $F_{\mathtt{base}} : \mathbf{Sig}_{\mathrm{id}} \to \mathbf{Set}$ is a functor with $F_{\mathtt{base}}(\Sigma) \subseteq \mathrm{for}(F_{\mathtt{sig}}(\Sigma))$, and
- $F_{\mathtt{for},\Sigma} : \mathrm{for}(\Sigma) \to \mathrm{for}(F_{\mathtt{sig}}(\Sigma))$ is a function for every signature Σ;

and the application of F on morphisms is given by $F(id_{\langle \Sigma, \Phi \rangle}) = id_{F(\Sigma, \Phi)}$.

Without a social life, the functor $F_{\mathtt{sig}}$ is equivalent to a function from signatures to signatures. It captures the effect of F on the signature, i.e., $F(\Sigma, \Phi)_1 = F_{\mathtt{sig}}(\Sigma)$. The functor $F_{\mathtt{base}}$ provides properties related to the signature elements. It is equivalent to a function from signatures to sets of axioms for the target signature. Finally, the function $F_{\mathtt{for},\Sigma}$ describes the effect of F on individual formulae $\varphi \in \Phi$. By abuse of notation we also write $F_{\mathtt{for},\Sigma}$ for the pointwise application of $F_{\mathtt{for},\Sigma}$ to a set of formulae Φ. In Section 3.4 we will see that our motivating examples can be formulated using STFs.

The family of functors presented in Example 1 fails to describe STFs for $\Sigma \neq \Sigma'$ and $\Sigma \neq \Sigma''$: the value of the resulting signature $T(\Sigma, \Phi)_1$ depends on Φ, i.e., it maps theories over the same signature to theories over different signatures, depending on the formulae.

STFs are well behaved with respect to set operations on collections of axioms. Furthermore, theories and STFs form a category.

Theorem 1 *(Additivity). Let $F : \mathbf{Th}_{\mathrm{id}} \to \mathbf{Th}_{\mathrm{id}}$ be a syntactic theory functor. The functor F is additive, i.e., $F(\Sigma, \Phi_1 \cup \Phi_2)_2 = F(\Sigma, \Phi_1)_2 \cup F(\Sigma, \Phi_2)_2$.*

Proof. First we note that the function $F_{\mathtt{for}}$ is additive for every signature Σ, i.e., $F_{\mathtt{for},\Sigma}(\Phi_1 \cup \Phi_2) = F_{\mathtt{for},\Sigma}(\Phi_1) \cup F_{\mathtt{for},\Sigma}(\Phi_2)$. Since $F_{\mathtt{for},\Sigma}$ is defined by its effect on each axiom this is clear. The claim is easily seen to hold by expanding the definition of both sides: $F(\Sigma, \Phi_1 \cup \Phi_2)_2 = F_{\mathtt{base}}(\Sigma) \cup F_{\mathtt{for},\Sigma}(\Phi_1 \cup \Phi_2) = F_{\mathtt{base}}(\Sigma) \cup F_{\mathtt{for},\Sigma}(\Phi_1) \cup F_{\mathtt{for},\Sigma}(\Phi_2) = F_{\mathtt{base}}(\Sigma) \cup F_{\mathtt{for},\Sigma}(\Phi_1) \cup F_{\mathtt{base}}(\Sigma) \cup F_{\mathtt{for},\Sigma}(\Phi_2) = F(\Sigma, \Phi_1)_2 \cup F(\Sigma, \Phi_2)_2$. □

We can interpret the identity functor $\mathrm{id}_{\mathbf{Th}_{\mathrm{id}}} : \mathbf{Th}_{\mathrm{id}} \to \mathbf{Th}_{\mathrm{id}}$ on objects by $\mathrm{id}_{\mathbf{Th}_{\mathrm{id}},\mathtt{sig}}(\Sigma) = \Sigma$, $\mathrm{id}_{\mathbf{Th}_{\mathrm{id}},\mathtt{base}}(\Sigma) = \emptyset$ and $\mathrm{id}_{\mathbf{Th}_{\mathrm{id}},\mathtt{for},\Sigma}(\Phi) = \Phi$, which taken together gives

$$\langle \mathrm{id}_{\mathbf{Th}_{\mathrm{id}},\mathtt{sig}}(\Sigma), \mathrm{id}_{\mathbf{Th}_{\mathrm{id}},\mathtt{base}}(\Sigma) \cup \mathrm{id}_{\mathbf{Th}_{\mathrm{id}},\mathtt{for},\Sigma}(\Phi) \rangle = \langle \Sigma, \emptyset \cup \Phi \rangle = \langle \Sigma, \Phi \rangle,$$

showing it is a syntactic theory functor.

Proposition 1 *(Composition of STFs). Given two STFs $F, G : \mathbf{Th}_{\mathrm{id}} \to \mathbf{Th}_{\mathrm{id}}$, their composition is a theory functor given by $F;G : \mathbf{Th}_{\mathrm{id}} \to \mathbf{Th}_{\mathrm{id}}$, with object components on $\langle \Sigma, \Phi \rangle$*

$$
\begin{aligned}
(F;G)_{\mathtt{sig}}(\Sigma) &= G_{\mathtt{sig}}(F_{\mathtt{sig}}(\Sigma)),\\
(F;G)_{\mathtt{base}}(\Sigma) &= G_{\mathtt{base}}(F_{\mathtt{sig}}(\Sigma)) \cup G_{\mathtt{for},F_{\mathtt{sig}}(\Sigma)}(F_{\mathtt{base}}(\Sigma)),\\
(F;G)_{\mathtt{for},\Sigma}(\Phi) &= G_{\mathtt{for},F_{\mathtt{sig}}(\Sigma)}(F_{\mathtt{for},\Sigma}(\Phi)).
\end{aligned}
$$

Proof. We need to show that this gives the functor $G(F(\Sigma, \Phi))$, which follows from the derivation

$$
\begin{aligned}
G(F(\Sigma,\Phi)) &= G(F_{\mathtt{sig}}(\Sigma), F_{\mathtt{base}}(\Sigma) \cup F_{\mathtt{for},\Sigma}(\Phi))\\
&= \langle G_{\mathtt{sig}}(F_{\mathtt{sig}}(\Sigma)), G_{\mathtt{base}}(F_{\mathtt{sig}}(\Sigma)) \cup G_{\mathtt{for},F_{\mathtt{sig}}(\Sigma)}(F_{\mathtt{base}}(\Sigma) \cup F_{\mathtt{for},\Sigma}(\Phi))\rangle\\
&= \langle G_{\mathtt{sig}}(F_{\mathtt{sig}}(\Sigma)), G_{\mathtt{base}}(F_{\mathtt{sig}}(\Sigma)) \cup G_{\mathtt{for},F_{\mathtt{sig}}(\Sigma)}(F_{\mathtt{base}}(\Sigma)) \cup G_{\mathtt{for},F_{\mathtt{sig}}(\Sigma)}(F_{\mathtt{for},\Sigma}(\Phi))\rangle\\
&= \langle (F;G)_{\mathtt{sig}}(\Sigma), (F;G)_{\mathtt{base}}(\Sigma) \cup (F;G)_{\mathtt{for},\Sigma}(\Phi)\rangle,
\end{aligned}
$$

as claimed. □

Theorem 2 *(Category of theories and STFs). There is a functor category with $\mathbf{Th}_{\mathrm{id}}$ as its object and STFs as morphisms.*

Proof. We know that STFs compose to STFs, and that the identity functor on theories is an STF. Since STFs are theory functors, composition is associative, and the identity theory functor is the identity for STF composition. This gives the claimed functor category. □

3.2. Remarks on the STF definition

Our definition of syntactic theory functors balances between many similar institution and related entailment system morphisms. We therefore look at some aspects of the STF definition which exposes some technical motivations. We use the terminology from [12].

An STF $F : \mathbf{Th}_{\mathrm{id}} \to \mathbf{Th}_{\mathrm{id}}$ is per definition a *signature preserving* endofunctor on $\mathbf{Th}_{\mathrm{id}}$, i.e., signatures are mapped to signatures without taking into consideration the formulae component of the theory. We can consider the functions $F_{\mathtt{for},\Sigma} : \mathrm{for}(\Sigma) \to \mathrm{for}(F_{\mathtt{sig}}(\Sigma))$ for every signature Σ, a natural transformation $F_{\mathtt{for}} : \mathrm{id}_{\mathbf{Sig}_{\mathrm{id}}}; \mathtt{for} \Rightarrow F_{\mathtt{sig}}; \mathtt{for}$. Then the STF F is $F_{\mathtt{for}}$-sensible: it is fully determined by $F(\Sigma, \emptyset)$ and $F_{\mathtt{for}}$.

These properties of STFs admit a more advanced meta-theory. In our current setting with discrete signature and discrete theory categories this does not immediately give much. In the setting of model consistent STFs that allows us to build institutions, see Section 5, many interesting results from [12] should carry over. We do not explore this general setting further here, but we provide some results for the structural properties of specific STFs in Section 5.2.

The STF notion is also similar to the notions of theoroidal (co)morphisms. While these morphisms allow for a more advanced institution, like the one of partial algebras, to be translated into a simpler institution, like the one of total algebras, these results cannot be applied directly to STFs. An STF is allowed to modify the model class in ways not compatible with the requirements for theoroidal maps. Further, STFs allow us to start in the simpler institution, conceptually map to a more complex institution and then do the encoding in the simpler institution. The example in Section 2.5 does this: it takes us from the total algebra of rings, adds a partial function inv, and then encodes this as a total algebra with error elements. Thus STFs have a strong flavour of being theoroidal without satisfying the criteria.

Codescu [13] works with generalised theoroidal comorphisms, which are similar to specification frame comorphisms. These also relate theories and models, but ignore the signature part. The purpose is to allow a minimalistic translation of theories to theories in order to reuse existing tools for one institution when solving problems in another institution. Such tools are better helped if the translation can be adapted to the actual set of formulas being transferred. This achieves the flexibility of translation as exposed in Example 1 above. Codescu's solution is thus deliberately not signature preserving. We believe signature preservation is significant for the purpose of STFs.

3.3. The kernel structuring mechanisms are STFs

Here, we consider the kernel language for structuring specifications in an arbitrary institution: presentation, union, renaming and hiding [14]. We present this in order to show that STFs cover the basic structuring mechanisms, but also to motivate why STFs are defined on the discrete theory category. Originally, [14] gives a model-semantics to its institution-independent structuring operations. However, as STFs concern syntax only, necessarily we have to work with a syntactic characterization. To this end we utilize the classical results concerning normalisation of structuring operations [15].

Presentations are already included in our setting. They are the finite theories: finite signatures with finite sets of finite axioms.

A *specification* is a syntactic object built from finite theories using structuring operations, i.e., a finite theory is a specification. The notation of a specification usually includes the pragmatic restriction to *finite* theories. In contrast, our discussion of STFs does not require this restriction. Its results also apply to infinite theories.

The union operator of specifications is a binary operator $\cup$. Given two specifications Sp_1 and Sp_2 over the same signature Σ for its normal form NF the following holds: $NF(Sp_1 \cup Sp_2) = (\Sigma, \Phi_1 \cup \Phi_2)$, where $NF(Sp_i) = (\Sigma, \Phi_i)$, $i = 1, 2$. Taking theories (Σ, Φ) and (Σ', Φ') rather than specifications, we can capture the union operation by a family of STFs indexed by theories $\langle \Sigma, \Phi \rangle$, i.e., $\text{union}_{\langle \Sigma, \Phi \rangle} : \mathbf{Th}_{\text{id}} \to \mathbf{Th}_{\text{id}}$,

$$\begin{aligned}
\text{union}_{\langle \Sigma, \Phi \rangle, \texttt{sig}}(\Sigma') &= \Sigma', \\
\text{union}_{\langle \Sigma, \Phi \rangle, \texttt{base}}(\Sigma') &= \begin{cases} \Phi & \text{when } \Sigma = \Sigma', \\ \emptyset & \text{otherwise,} \end{cases} \\
\text{union}_{\langle \Sigma, \Phi \rangle, \texttt{for}, \Sigma'}(\Phi') &= \Phi'.
\end{aligned}$$

Since the union operator only works when both argument theories have the same signature, the functor gets a 'bump' when they are equal, and is otherwise the identity functor. This is OK for a functor on the discrete theory category, but will cause problems if the theories are connected with signature morphisms. The 'bump' can be smoothed if the underlying institution, e.g., allows union on signatures, but such a smoother union STF will only work for such institutions and not be an institution-independent solution.

Writing $=_{\text{mod}}$ for equality of model classes, we can relate union of specifications to our STF.

Proposition 2. $(\Sigma, \Phi_1) \cup (\Sigma, \Phi_2) =_{\text{mod}} \text{union}_{\langle \Sigma, \Phi_1 \rangle}(\Sigma, \Phi_2) =_{\text{mod}} \text{union}_{\langle \Sigma, \Phi_2 \rangle}(\Sigma, \Phi_1)$.

Proof. $(\Sigma, \Phi_1) \cup (\Sigma, \Phi_2) = (\Sigma, \Phi_1 \cup \Phi_2) = \text{union}_{\langle \Sigma, \Phi_1 \rangle}(\Sigma, \Phi_2) = \text{union}_{\langle \Sigma, \Phi_2 \rangle}(\Sigma, \Phi_1)$. □

Renaming applies a signature morphism to a specification. Given a specification Sp and a signature morphism $\sigma : \Sigma \to \Sigma'$, for its normal form $NF(Sp) = (\Sigma, \Phi)$ the following holds: $NF(\text{rename } Sp \text{ by } \sigma : \Sigma \to \Sigma') = (\Sigma', \text{for}(\sigma : \Sigma \to \Sigma')(\Phi))$. We can capture the renaming operation by a family of STFs indexed by signature morphisms $\sigma : \Sigma \to \Sigma'$, i.e., $\text{rename}_{\sigma : \Sigma \to \Sigma'} : \mathbf{Th}_{\text{id}} \to \mathbf{Th}_{\text{id}}$,

$$\begin{aligned}
\text{rename}_{\sigma : \Sigma \to \Sigma', \texttt{sig}}(\Sigma'') &= \begin{cases} \Sigma' & \text{when } \Sigma = \Sigma'', \\ \Sigma'' & \text{otherwise,} \end{cases} \\
\text{rename}_{\sigma : \Sigma \to \Sigma', \texttt{base}}(\Sigma'') &= \emptyset, \\
\text{rename}_{\sigma : \Sigma \to \Sigma, \texttt{for}, \Sigma''}(\Phi'') &= \begin{cases} \text{for}(\sigma : \Sigma \to \Sigma')(\Phi'') & \text{when } \Sigma = \Sigma'', \\ \Phi'' & \text{otherwise,} \end{cases}
\end{aligned}$$

Here it is quite clear that the 'bump' at Σ will cause problems when interacting with the signature morphisms in a non-discrete theory category. Further note that we get an STF for each signature morphism, somehow 'reconstructing' the structure in the category of signatures we removed. With this rename STF, we obtain:

Proposition 3. *rename* (Σ, Φ) *by* $\sigma : \Sigma \to \Sigma' =_{\text{mod}} \text{rename}_{\sigma : \Sigma \to \Sigma'}(\Sigma, \Phi)$.

Proof. $\text{rename}(\Sigma, \Phi) \text{ by } \sigma : \Sigma \to \Sigma' = (\Sigma', \text{for}(\sigma : \Sigma \to \Sigma')(\Phi)) = \text{rename}_{\sigma : \Sigma \to \Sigma'}(\Sigma, \Phi)$. □

Hiding fails to be a syntactic structuring mechanism in many logics [16], i.e., there are specifications formed with hiding for which there does not exist a specification without hiding which is semantically equivalent, see e.g. [17]. Consequently, we refrain from characterizing hiding as an STF.

Both the union and rename STFs are institution-independent, i.e., can be utilized in any institution-based specification formalism.

3.4. STFs for our motivating examples

All of our motivating examples give rise to STFs. They are CASL specific. As $\mathbf{Th}_{\text{id}}$ and $\mathbf{Sig}_{\text{id}}$ are discrete categories, it suffices to specify the effect on objects only. We are now working in the CASL institution which allows union of signatures.

Concerning finite sets of distinct constants, cf. Section 2.1, we define for a signature $\Sigma = (S, F)$ consisting of a set of sort symbols S and a set of function symbols F and for each formula φ over Σ:

$$\begin{aligned}
\text{free}_{\langle s, \{c_1, \ldots, c_n\} \rangle, \texttt{sig}}(S, F) &= (S \cup \{s\}, F \cup \{c_1 : s, \ldots, c_n : s\}), \\
\text{free}_{\langle s, \{c_1, \ldots, c_n\} \rangle, \texttt{base}}(S, F) &= \{c_i \neq c_j \mid 1 \leq i < j \leq n\}, \\
\text{free}_{\langle s, \{c_1, \ldots, c_n\} \rangle, \texttt{for}, \langle S, F \rangle}(\varphi) &= \varphi.
\end{aligned}$$

Here, s is a sort name and $c_1, \ldots, c_n, n \geq 0$, are constant names. Note that the free functor actually is a family of functors thanks to its parametrisation. With this STF, we can, e.g., write

$$\text{free}_{s,\{c1,c2,c3,c4,c5\}}(\{\}),$$

where {} denotes the empty CASL specification. This "free" specification has the same model class as SORTWITHMINIMUM5ELEMENTS.

Similarly, we can define an STF generated, where

$$\begin{aligned}
\text{generated}_{\langle s,\{c_1,\ldots,c_n\}\rangle,\texttt{sig}}(S,F) &= (S \cup \{s\}, F \cup \{c_1 : s, \ldots, c_n : s\}),\\
\text{generated}_{\langle s,\{c_1,\ldots,c_n\}\rangle,\texttt{base}}(S,F) &= \{\forall x : s \bullet x = c_1 \vee \ldots \vee x = c_n\},\\
\text{generated}_{\langle s,\{c_1,\ldots,c_n\}\rangle,\texttt{for},\langle S,F\rangle}(\varphi) &= \varphi
\end{aligned}$$

With this STF, we obtain

$$\text{SORTWITHMAXIMUM5ELEMENTS} =_{\text{mod}} \text{generated}_{s,\{c1,c2,c3,c4,c5\}}(\{\}).$$

Concerning the Leibniz rule, cf. Section 2.3, it is necessary to modify formulae within the `for` component:

$$\begin{aligned}
\text{augment}_{\langle dir,D\rangle,\texttt{sig}}(S,F) &= \langle S \cup \{dir\}, F \setminus D \cup \{aug_{\texttt{sig}}(\langle S,F\rangle, dir, d) \mid d \in D \cap F\}\rangle,\\
\text{augment}_{\langle dir,D\rangle,\texttt{base}}(S,F) &= \emptyset,\\
\text{augment}_{\langle dir,D\rangle,\texttt{for},\langle S,F\rangle}(\varphi) &= aug_{\texttt{for}}(\langle S,F\rangle, dir, D, \varphi),
\end{aligned}$$

where dir is a sort symbol and D is a set of operations. The idea is that $D \subseteq F$ is a set of existing operations we want to augment with an extra argument. We further define:

$$aug_{\texttt{sig}}(\langle S,F\rangle, dir, d : s_1 \times \cdots \times s_k \to t) = d : dir \times s_1 \times \cdots \times s_k \to t,$$

$$aug_{\texttt{for}}(\langle S,F\rangle, dir, D, \varphi) = \begin{cases} \varphi & ;\ \varphi \text{ has no term in } d \in D \\ \forall x : dir \bullet \varphi' & ;\ \text{otherwise} \end{cases}$$

where x is a fresh variable over φ, and φ' is like φ but with each term $d(t_1, \ldots, t_k)$ replaced by $d(x, t_1, \ldots, t_k)$, if $d \in D$. With this STF, we obtain

$$\text{DIRECTIONALLEIBNIZRULE} =_{\text{mod}} augment_{\langle dir,\{D:r\to r\}\rangle}(\text{LEIBNIZRULE}).$$

Next we consider the example of lifting, cf. Section 2.4.

$$\begin{aligned}
\text{lift}_{\langle A,I,E\rangle,\texttt{sig}}(S,F) &= (S \cup \{A, I, E\}, F \cup \{get : A \times I \to E\}\\
&\qquad \cup \{o : A \times \ldots \times A \to A \mid o : E \times \ldots \times E \to E \in F\}),\\
\text{lift}_{\langle A,I,E\rangle,\texttt{base}}(S,F) &= \{\forall a, b : A \bullet a = b \Leftrightarrow \forall i : I \bullet get(a,i) = get(b,i)\}\\
&\qquad \cup \{\forall a_1, \ldots, a_k : A; i : I \bullet\\
&\qquad\quad get(o(a_1, \ldots, a_k), i) = o(get(a_1, i), \ldots, get(a_k, i))\\
&\qquad\quad \mid o : E \times \ldots \times E \to E \in F\},\\
\text{lift}_{\langle A,I,E\rangle,\texttt{for},\langle S,F\rangle}(\varphi) &= \varphi,
\end{aligned}$$

where A is the 'array' sort, to which operations shall be lifted, I is the index sort of A, and E is the sort from which operations should be lifted. An alternative version could also lift equational axioms on E to equational axioms on A by providing the conjunction of the original equation on E with the similarly formulated axiom on A when defining $\text{lift}_{\langle A,I,E\rangle,\texttt{for},\langle S,F\rangle}(\varphi)$.

Finally we capture the rule for introducing error elements to a total specification. Here we add an error element for each type S, add new axioms for error propagation for each function declaration in F, and protect all existing axioms against inadvertently applying to error elements. This gives an STF error : $\mathbf{Th}_{\text{id}} \to \mathbf{Th}_{\text{id}}$,

$$\begin{aligned}
\text{error}_{\texttt{sig}}(S,F) &= \langle S, F \cup \{s_{\text{error}} : s \mid s \in S\}\rangle,\\
\text{error}_{\texttt{base}}(S,F) &= \{\forall x_1 : s_1, \ldots, x_k : s_k \bullet f(x_1, \ldots, s_{i,\text{error}} \ldots, x_k) = s_{\text{error}}\\
&\qquad \mid (f : s_1, \ldots, s_k \to s) \in F, i = 1, \ldots, k\},\\
\text{error}_{\texttt{for},\langle S,F\rangle}(\varphi) &= \varphi',
\end{aligned}$$

where φ' is the same axiom as φ, but where the axiom has a conditional $x_j \neq s_{j,\text{error}}$ for every variable $x_j : s_j$ used in the axiom.[6]

Based on the successful representation of all of our motivating examples, we conclude this section with the conjecture that STFs are a concept broad enough to capture design patterns for specifications.

4. STFs are syntactic structuring operations

Most specification languages provide structuring mechanisms going beyond expressing specifications simply as, often finite,[7] theories $T = \langle \Sigma, \Phi \rangle$. This gives a structured specification language with $\mathit{Struct}(\mathbf{Th}_{\text{id}})$ as the set of all specifications that we can provide.

A structuring language $\mathit{Struct}(\mathbf{Th}_{\text{id}})$ is *syntactic* over $\mathbf{Th}_{\text{id}}$ [16] if for every structured specification $Sp \in \mathit{Struct}(\mathbf{Th}_{\text{id}})$ there exists a theory $T \in \mathbf{Th}_{\text{id}}$ such that

$$\text{mod}(Sp) = \text{mod}(T),$$

i.e., the structuring mechanisms do not offer more expressive power. They rather support the stepwise construction of specifications and help in understanding large specifications by imposing structure on them. Often the relation between a specification Sp and its equivalent theory T can be expressed by a function

$$\text{flatten} : \mathit{Struct}(\mathbf{Th}_{\text{id}}) \to \mathbf{Th}_{\text{id}}$$

where $\text{mod}(Sp) = \text{mod}(\text{flatten}(Sp))$. Usually, $\mathit{Struct}(\mathbf{Th}_{\text{id}})$ includes presentations $\iota : \mathbf{Th}_{\text{id}} \to \mathit{Struct}(\mathbf{Th}_{\text{id}})$ with semantics $\text{mod}(\iota(T)) = \text{mod}(T)$.

The set of syntactic structuring mechanisms of a specification language can be extended by STFs:

Definition 4. Let *Struct* be a set of syntactic structuring mechanisms with a flatten operator, let $F : \mathbf{Th}_{\text{id}} \to \mathbf{Th}_{\text{id}}$ be an STF, let $\bar{F} \notin \mathit{Struct}$ be a new symbol that shall denote the application of F to a specification. Define $\mathit{Struct}_{\bar{F}} = \mathit{Struct} \cup \{\bar{F}\}$. We extend the flattening operation by defining $\text{flatten}' : \mathit{Struct}_{\bar{F}}(\mathbf{Th}_{\text{id}}) \to \mathbf{Th}_{\text{id}}$

$$\text{flatten}'(Sp) = \begin{cases} \langle \Sigma, \Phi \rangle & \text{when } Sp = \iota(\Sigma, \Phi), \\ F(\text{flatten}'(Sp')) & \text{when } Sp = \bar{F}(Sp'), \\ \text{flatten}(G(\text{flatten}'(Sp'))) & \text{when } Sp = G(Sp') \text{ for } G \in \mathit{Struct} \setminus \{\iota\}, \end{cases}$$

and define the model functor by

$$\text{mod}(\bar{F}(Sp)) = \text{mod}(F(\text{flatten}'(Sp))).$$

For a collection of theory functors $\mathcal{F}$, we obtain a syntactic structuring language $\mathit{Struct}_{\mathcal{F}}(\mathbf{Th}_{\text{id}})$:

Theorem 3. *Let Struct be a set of syntactic structuring mechanisms with an operation* $\text{flatten} : \mathit{Struct}(\mathbf{Th}_{\text{id}}) \to \mathbf{Th}_{\text{id}}$. *Let* $\mathcal{F}$ *be a set of STFs. Define* $\mathit{Struct}_{\mathcal{F}} = \mathit{Struct} \cup \{\bar{F} \mid F \in \mathcal{F}\}$. *Then* $\mathit{Struct}_{\mathcal{F}}$ *is syntactic over* $\mathbf{Th}_{\text{id}}$.

Proof. (by induction on the terms of the structuring language) Given a theory T, we have by definition $\text{mod}(\iota(T) = \text{mod}(T)$. Consider $s(Sp_1, \ldots, Sp_k)$, $k \geq 1$, where $s \in \mathit{Struct}$ and $Sp_i \in \mathit{Struct}_{\mathcal{F}}$, $i = 1, \ldots, k$. Then by I.H., there exist theories $T_1, \ldots, T_k \in \mathbf{Th}_{\text{id}}$ with $\text{mod}(Sp_i) = \text{mod}(T_i)$, $i = 1, \ldots, k$. As $s \in \mathit{Struct}$, we have $\text{flatten}(s(T_1, \ldots, T_k)) \in \mathbf{Th}_{\text{id}}$ and $\text{mod}(s(Sp_1, \ldots, Sp_k)) = \text{mod}(\text{flatten}(s(T_1, \ldots, T_k)))$. Consider $\bar{F}(Sp)$, where $F \in \mathcal{F}$ and $Sp \in \mathit{Struct}_{\mathcal{F}}$. Then by I.H., there exists a theory T with $\text{mod}(Sp) = \text{mod}(T)$. By Definition 4 we have $\text{flatten}(\bar{F}(T)) = F(T) \in \mathbf{Th}_{\text{id}}$ and $\text{mod}(\bar{F}(Sp)) = \text{mod}(\text{flatten}(\bar{F}(T)))$. □

A potential critisism to STFs is that they work on discrete signature categories and thus ignore part of the structure given through an institutional setting. Thus, one should note that the moment one works with $\mathit{Struct}_{\mathcal{F}}(\mathbf{Th}_{\text{id}})$, where *Struct* includes the renaming operation, the morphisms are back again:

Remark 1. Note that with the renaming STF, on the structuring level we get the signature morphisms back: any signature morphism σ gives rise to an STF rename_{σ}.

[6] Note this is a bit simplistic and only works well for axioms in the form of boolean expressions, which includes equational and conditional equational axioms. Axioms with embedded quantifiers need a more advanced treatment.

[7] Both, signature and set of formulae are finite.

4.1. Case study: remote control

Coming back to our initial example of specifying a remote control, e.g., for a TV, we demonstrate how STFs can ease the development of a specification for such a device.

Abstractly seen, a remote control provides a user interface of several buttons. When a user presses a button, the remote control sends out an (infrared) signal. In remotes for entertainment, this signal is a bit string consisting of three substrings: the first string is an identifier of the company, the second string encodes the type of the device (e.g., TV, DVD, game console, satellite receiver), the third string finally is a code specific to the pressed button. Using a loose specification, this first view on a remote control can be made precise in CASL:

spec REMOTECONTROL =
  sorts *Button*, *Signal*
  ops *companyId*, *deviceType* : *Signal*;
    codeOf : *Button* → *Signal*;
    buttonCode : *Button* → *Signal*;
    __++__ : *Signal* × *Signal* → *Signal*, *assoc*
  ∀ *b* : *Button* • *codeOf*(*b*) = (*companyId* ++ *deviceType*) ++ *buttonCode*(*b*)
end

Thinking of Boolean strings as the intended carrier set of *Signal*, in the intended model __++__ stands for string concatenation.

Following the paradigm of step-wise refinement, such abstract view on remote controls in home entertainment can be made more concrete. For easy selection of channels, a remote control shall have at least 10 different buttons for the digits 0 to 9. Furthermore, there should be a button to turn the device on and off. All these buttons should be different. Using our STF *free* we can specify:

spec REMOTECONTROLWITHELEVENDIFFERENTBUTTONS =
  $free_{\langle Button, \{b_0, b_1, b_2, b_3, b_4, b_5, b_6, b_7, b_8, b_9, b_{onoff}\}\rangle}$ (REMOTECONTROL)
end

Often, a remote control shall control several devices. To this end remote controls work in different modes. Each mode corresponds to a device, i.e., the second string of the signal is mode dependent. Using our *augment* STF, we can add this mode dependency by turning the constant *deviceType* into a function:

spec REMOTECONTROLWITHMODES =
  $augment_{\langle Mode, \{deviceType\}\rangle}$ (REMOTECONTROLWITHELEVENDIFFERENTBUTTONS)
end

In a next step, we might want to specify that the remote control shall be able to deal at least with TV, DVD, and a SAT (receiver). Here, we can again use our STF *free* in order to add these as different modes:

spec REMOTECONTROLWITHSPECIFICMODES =
  $free_{\langle Mode, \{TV, DVD, SAT\}\rangle}$ (REMOTECONTROLWITHMODES)
end

Finally, one might want to add that there shall be two buttons “up” and “dn” for the purpose of volume control:

spec REMOTECONTROLWITHVOLUMECONTROL =
  $free_{\langle Button, \{b_0, b_1, b_2, b_3, b_4, b_5, b_6, b_7, b_8, b_9, b_{onoff}, up, dn\}\rangle}$ (REMOTECONTROLWITHSPECIFICMODES)
end

Note that the STF *free* specifies a minimal number of elements in the carrier set. Thus, this second application of the STF free to the sort *Button* is consistent, as it simply increases this minimal number. That the STF doubles certain axioms, such as, e.g., $\neg b_0 = b_1$, is unproblematic.

Flattening this specification, we obtain

spec FLATTENEDREMOTECONTROLWITHVOLUMECONTROL =
  sorts *Button*, *Signal*, *Mode*
  ops *companyId* : *Signal*;

$deviceType : Mode \to Signal$;
$codeOf : Button \to Signal$;
$buttonCode : Button \to Signal$;
$\_\!+\!+\!\_ : Signal \times Signal \to Signal$, *assoc*;
$TV, DVD, SAT : Mode$
$b_0, b_1, b_2, b_3, b_4, b_5, b_6, b_7, b_8, b_9, b_{onoff}, up, dn : Button$
$\forall\ m : Mode;\ b : Button \bullet codeOf(b) = (companyId\ ++\ deviceType(m))\ ++\ buttonCode(b)$
…(distinctiveness axioms) …

end

where we leave out the 81 distinctiveness axioms: 78 are needed to express that there are at least 13 distinct buttons, 3 are needed to specify that there are at least 3 different modes. One could argue that these axioms are 'primitive'. However, note they cannot be written any shorter in CASL: CASL's free type construct is semantically different; it would specify that there are exactly 13 different buttons and exactly 3 different modes. Furthermore, as the CASL free construct is not in first order logic, using it in specifications requires specific theorem proving support.

It is important to note that this specification exercise can be re-factored. The order in which we applied the various STFs plays (at least in this example) no role. Thus, one could first build up the infrastructure for modes in two steps, and only then, in a third step, care about the buttons:

spec REMOTECONTROLREFACTORED =
$free_{\langle Button, \{b_0, b_1, b_2, b_3, b_4, b_5, b_6, b_7, b_8, b_9, b_{onoff}, up, dn\}\rangle}$
($free_{\langle Mode, \{TV, DVD, SAT\}\rangle}$
($augment_{\langle Mode, \{deviceType\}\rangle}$ (REMOTECONTROL)))

end

Without studying this topic further, we mention that this refactoring exercise illustrates that STFs come with algebraic properties such as subsumption or commutativity of STF application.

4.1.1. Methodological analysis of the case study

From a methodological point of view, our case study demonstrates a number of advantages through using STFs:

- STFs can concisely express connections between specifications that cannot be expressed otherwise. Without using the *augment* STF as a specification transformation, it would not be possible to derive the specification REMOTECONTROLWITHMODES from the specification REMOTECONTROLWITHELEVENDIFFERENTBUTTONS – although this appears to be a 'natural' development step.
- Using a number of STFs in sequence is a way to enrich specifications to capture more and more aspects of a system.
- STFs allow for shorter and more concise specification text.
- STFs can keep the specification logic simple, allowing for better theorem proving support.
- The effect of STFs is controllable. Through flattening, the specifier can find out if the application of an STF yields the desired result. This transparency allows for faithful modelling, where specification elements are traced back to a given narrative.

4.2. How to make STFs available to specifiers?

The theoretical framework that STFs provide has the potential to greatly support the specifier. This, however, is contingent on proper tool support. Though the formalism is still at an early stage, certain elements of such tools are already clear. They should support:

Application of STFs. Tools need to support the application of STFs. This support will essentially be of type-checking nature. It will ensure that all parameters of an STF are instantiated, e.g., the above free STF requires two parameters, one sort symbol and one set of constant symbols. Furthermore, it will ensure that conditions linking actual parameters of an STF and a specification are fulfilled, e.g., in the above augment STF its second parameter D should be a subset of the operation symbols F of the specification.

Flattening. Tools need to be able to flatten STFs and show the result to the specifier. The flattening operation should allow the specifier to select which of the possibly many STF applications to be flattened.

Libraries of STFs. Tools need to offer access to different libraries of STFs. These libraries will fall into different categories: some STFs are of general nature, as, e.g., the specification structuring operations or the constructions in Cerioli's and Reggio's study on the interplay between partiality, subsorting and predicates [3]; other STFs will be of domain-specific nature, i.e., they will capture constructions identified by domain engineering.

Definition of new STFs STFs gain flexibility through two elements: the shape of their parameters is unrestricted; the functions that they realise are unrestricted. This makes it hard to identify 'standard constructs' that would allow one

to build 'all' STFs. However, tools should define an abstract interface for STFs. Such an interface will require the implementation for three component functions. It will also ensure that the component functions only make use of those parts of the specification that should be accessible to them; i.e., only signatures are passed to $F_{\mathtt{sig}}$ and $F_{\mathtt{base}}$, while $F_{\mathtt{for}}$ depends on both, signature and formulae of a theory.

5. Building institutions from STFs

When building complex specifications from simpler ones, we are interested in knowing what properties are valid for the new specification and how it relates to the components' properties. For union we know that all properties deducible in each component hold for the combined specification. This holds even if the component specifications contradict each other, in which case the combined specification has no models. Similar results exist for renaming.

In this section we study constructions and property transport between combined and component specifications for STFs. First we look at building syntactic structure from STFs and the property of model consistency. Then we explore how our motivating example STFs transport properties. Finally, we use this insight to create more complex institutions from base institutions with model consistent STFs.

5.1. Model consistent STFs

Since STFs are signature preserving we can use that to equip our discrete signature category with morphisms.

Definition 5 *(STF induced arcs on signatures).* Let $F : \mathbf{Th}_{\mathrm{id}} \to \mathbf{Th}_{\mathrm{id}}$ be an STF, let Σ be a signature. Then F defines an arc to another signature, namely $\Sigma_F : \Sigma \to F_{\mathtt{sig}}(\Sigma)$. We call an arc Σ_F *trivial* whenever for all $\Phi \subseteq \mathrm{for}(\mathbf{Sig})$ it holds that $F(\Sigma, \Phi) = \langle \Sigma, \Phi \rangle$, non-trivial otherwise.

Note that for an arc $\Sigma_F : \Sigma \to \Sigma$ it depends on the effect that F has on formulae whether Σ_F is trivial or non-trivial. In a trivial arc, $F_{\mathtt{sig}}(\Sigma) = \Sigma$, $F_{\mathtt{base}}(\Sigma) = \emptyset$, and $F_{\mathtt{for},\Sigma}(\Phi) = \Phi$, for all Φ.

Definition 6 *(Model consistent STF).* Let $\mathcal{INST} = \langle \mathbf{Sig}, \mathrm{for} : \mathbf{Sig} \to \mathbf{Set}, \mathrm{mod} : \mathbf{Sig}^{\mathrm{op}} \to \mathbf{CAT}, \models \rangle$ be an institution with associated discrete theory category $\mathbf{Th}_{\mathrm{id}}$.

An STF $F : \mathbf{Th}_{\mathrm{id}} \to \mathbf{Th}_{\mathrm{id}}$ is *model consistent* if, for every $\Sigma_F : \Sigma \to F_{\mathtt{sig}}(\Sigma)$, there is a functor $\mu_{\Sigma_F} : \mathrm{mod}(\Sigma) \leftarrow \mathrm{mod}(F_{\mathtt{sig}}(\Sigma))$ such that for all $M' \in \mathrm{mod}(F_{\mathtt{sig}}(\Sigma))$ and all $\varphi \in \mathrm{for}(\Sigma)$ the condition

$$\mu_{\Sigma_F}(M') \models_{\Sigma} \varphi \iff M' \models_{F_{\mathtt{sig}}(\Sigma)} F(\Sigma, \{\varphi\})_2 \tag{2}$$

holds.

Note the 'asymmetry' in the equivalence (2): there is a single formula on the lhs of the equivalence, while there possibly is a set of formulae on the rhs of the equivalence. Further note that only non-trivial arcs are interesting for deciding model consistency.

Proposition 4 *(Trivial arcs generate model consistent functors). Let $\Sigma_F : \Sigma \to \Sigma$ be a trivial arc for an STF F. Then $\mu_{\Sigma_F} = id_{\mathrm{mod}(\Sigma)}$ is model consistent.*

Proof. Let $\varphi \in \mathtt{for}(\Sigma)$ be a formula and $M' \in \mathrm{mod}(F_{\mathtt{sig}}(\Sigma)) = \mathrm{mod}(\Sigma)$ be a model. Then $\mu_{\Sigma_F}(M') = M'$, hence (2) reduces to $M' \models_{\Sigma} \varphi \iff M' \models_{F_{\mathtt{sig}}(\Sigma)} F(\Sigma, \{\varphi\})_2$, which trivially holds since $F(\Sigma, \{\varphi\})_2 = \langle \Sigma, \{\varphi\} \rangle_2 = \{\varphi\}$. □

In Definition 6 we have chosen $\mu_{\Sigma_F} : \mathrm{mod}(\Sigma) \leftarrow \mathrm{mod}(F_{\mathtt{sig}}(\Sigma))$ to be a functor, i.e., μ_{Σ_F} 'transports' both, objects and morphisms, of the model category. However, model morphisms do not play a role in equivalence (2). Thus, alternatively, one could also work with a discrete model category. There are STFs F that allow the definition of μ_{Σ_F} only in discrete model categories – see the discussion of the STF augment in Section 5.2.1 below. The constructions of an STF Path institution and an STF institution in Section 5.3 work also when one replaces the model categories with discrete ones.

Proposition 5. *The composition of model consistent STFs is a model consistent STF.*

Proof. Let F and G be model consistent STFs with related model functors $\mu_{\Sigma_F} : \mathrm{mod}(\Sigma) \leftarrow \mathrm{mod}(F_{\mathtt{sig}}(\Sigma))$ and $\mu_{F_{\mathtt{sig}}(\Sigma)_G} : \mathrm{mod}(F_{\mathtt{sig}}(\Sigma)) \leftarrow \mathrm{mod}(G_{\mathtt{sig}}(F_{\mathtt{sig}}(\Sigma)))$, respectively. Then the model functor $\mu_{\Sigma_F} \circ \mu_{\Sigma_G}$ will be model consistent with $F; G$. Let $N' \in \mathrm{mod}(F_{\mathtt{sig}}(\Sigma))$ and $\varphi \in \mathtt{for}(\Sigma)$, and $M' \in \mathrm{mod}(G_{\mathtt{sig}}(F_{\mathtt{sig}}(\Sigma)))$ and $\psi \in \mathtt{for}(F_{\mathtt{sig}}(\Sigma))$, then

$$\begin{aligned} \mu_{\Sigma_F}(N') \models_{\Sigma} \varphi &\iff N' \models_{F_{\mathtt{sig}}(\Sigma)} F(\Sigma, \{\varphi\})_2, \\ \mu_{\Sigma_G}(M') \models_{F_{\mathtt{sig}}(\Sigma)} \psi &\iff M' \models_{G_{\mathtt{sig}}(F_{\mathtt{sig}}(\Sigma))} G(F_{\mathtt{sig}}(\Sigma), \{\psi\})_2. \end{aligned}$$

By inserting $\mu_{\Sigma_G}(M')$ for N' and $F(\Sigma, \{\varphi\})_2$ for ψ we get

$$\mu_{\Sigma_F}(\mu_{\Sigma_G}(M')) \models_\Sigma \varphi \iff \mu_{\Sigma_G}(M') \models_{F_{\mathtt{sig}}(\Sigma)} F(\Sigma, \{\varphi\})_2,$$
$$\mu_{\Sigma_G}(M') \models_{F_{\mathtt{sig}}(\Sigma)} F(\Sigma, \{\varphi\})_2 \iff M' \models_{G_{\mathtt{sig}}(F_{\mathtt{sig}}(\Sigma))} G(F_{\mathtt{sig}}(\Sigma), F(\Sigma, \{\varphi\})_2)_2,$$

which simplifies to

$$\begin{aligned}(\mu_{\Sigma_F} \circ \mu_{\Sigma_G})(M') \models_\Sigma \varphi &\iff M' \models_{G_{\mathtt{sig}}(F_{\mathtt{sig}}(\Sigma))} G(F(\Sigma, \{\varphi\})_1, F(\Sigma, \{\varphi\})_2)_2,\\ &\iff M' \models_{G_{\mathtt{sig}}(F_{\mathtt{sig}}(\Sigma))} G(F(\Sigma, \{\varphi\}))_2,\\ &\iff M' \models_{(F;G)_{\mathtt{sig}}(\Sigma)} (F;G)(\Sigma, \{\varphi\}))_2. \quad \square\end{aligned}$$

Corollary 1. *There is a functor category with* $\mathbf{Th}_{\mathrm{id}}$ *as object and model consistent STFs as morphisms. It is a subcategory of the category of theories and STFs.*

Proof. The identity functor is a model consistent STF (it has only trivial arcs), and model consistent STFs compose, see Proposition 5. $\square$

Naturally, not all STFs are model consistent. We can make two general observations on having and not having model consistency. The first is a positive result relating model consistency to signature morphisms in the underlying institution. The second is a negative result for model consistency of STFs with non-trivial base axioms.

Proposition 6 *(Model consistency and signature morphisms). Let F be a renaming STF for a signature morphism $\sigma : \Sigma \to \Sigma'$. Then F is model consistent.*

Proof. The renaming STF defines trivial arcs everywhere except at Σ while at Σ it behaves like the signature morphism. In the former case the identity model functor provides model consistency. In the latter case the reduct functor $\mathrm{mod}(\sigma^{\mathrm{op}})$ provides model consistency. Thus for every signature the STF F is model consistent. $\square$

Proposition 6 easily extends to families of STFs defining some subclass of signature morphisms. For instance in CASL this could be STFs defining signature inclusion only, STFs defining name changes only, STFs defining swapping of arguments only, etc. This does not preclude STFs not being related to existing signature morphisms also being model consistent.

Proposition 7 *(Model consistency and non-trivial base axioms). Let Σ be a signature for which tautologies exist and let F be an STF where $F_{\mathtt{base}}(\Sigma)$ is non-trivial, i.e., includes a formula which is not a tautology. Then F is not model consistent.*

Proof. Let φ be a trivial axiom for Σ. By the assumption there exists an $M' \in \mathrm{mod}(F_{\mathtt{sig}}(\Sigma))$ such that $M' \notin \mathrm{mod}(F_{\mathtt{base}}(\Sigma))$. Then any functor $\mu_{\Sigma_F} : \mathrm{mod}(\Sigma) \leftarrow \mathrm{mod}(F_{\mathtt{sig}}(\Sigma))$ will have that $\mu_{\Sigma_F}(M') \models_\Sigma \varphi$ while $M' \not\models_{F_{\mathtt{sig}}(\Sigma)} F(\Sigma, \{\varphi\})_2$. Hence no model functor can make F model consistent. $\square$

In spite of not being model consistent, STFs may still transport formulae in one or the other direction of (2). Since most of our example STFs add non-trivial base axioms the negative result for such STFs motivates a slightly weaker notion for transporting formulae, which considers only models of the theory $(F_{\mathtt{sig}}(\Sigma), F_{\mathtt{base}}(\Sigma))$ rather than all models of $F_{\mathtt{sig}}(\Sigma)$:

Definition 7 *(Formula transportation by STF).* The STF F *transports formulae* when for all signatures Σ there is a functor $\mu_{\Sigma_F} : \mathrm{mod}(\Sigma) \leftarrow \mathrm{mod}(F_{\mathtt{sig}}(\Sigma), F_{\mathtt{base}}(\Sigma))$ such that for each model $M' \in \mathrm{mod}(F_{\mathtt{sig}}(\Sigma), F_{\mathtt{base}}(\Sigma))$ and for each formula $\varphi \in \mathrm{for}(\Sigma)$ one of

$$\mu_{\Sigma_F}(M') \models_\Sigma \varphi \Longrightarrow M' \models_{F_{\mathtt{sig}}(\Sigma)} F_{\mathtt{for},\Sigma}(\varphi), \tag{3}$$
$$\mu_{\Sigma_F}(M') \models_\Sigma \varphi \Longleftarrow M' \models_{F_{\mathtt{sig}}(\Sigma)} F_{\mathtt{for},\Sigma}(\varphi), \tag{4}$$

holds. Holding with "$\Longrightarrow$", equation (3), is *preserving* formulae, and holding with "$\Longleftarrow$", equation (4), is *reflecting* formulae.

Proposition 8. *If an STF is model consistent it transports formulae both ways.*

Proof. For an STF F and signature Σ we know that $F(\Sigma, \{\varphi\})_2 = F_{\mathtt{base}}(\Sigma) \cup \{F_{\mathtt{for},\Sigma}(\varphi)\}$. Hence the transport requirements both ways will hold for all $M' \in \mathrm{mod}(F_{\mathtt{sig}}(\Sigma))$, specifically for all $M' \in \mathrm{mod}(F_{\mathtt{sig}}(\Sigma), F_{\mathtt{base}}(\Sigma)) \subseteq \mathrm{mod}(F_{\mathtt{sig}}(\Sigma))$ as required for transport of formulae both ways. $\square$

	free	generated	lift	union	rename	error	augment′	augment
model consistent	–	–	–	–	✓	–	–	–
preserve	✓	✓	✓[a]	✓	✓	(✓)[b]	–	–
reflect	✓	✓	✓	✓	✓	–	✓	(✓)[c]

Fig. 1. Reflection and preservation properties for STFs in the context of CASL. [a]In addition lifts equational axioms from elements to arrays. [b]Only for boolean expression axioms. [c]Only for discrete model categories.

Thus renaming STFs both preserve and reflect formulae. More generally, an STF translates formulae both ways when the formulae mapping component of an STF is compatible with a signature morphism.

Proposition 9. *Let F be an STF and $\Sigma_F : \Sigma \to \Sigma'$ be a signature morphism in the underlying institution for each Σ_F such that $F_{\mathrm{for},\Sigma} = \mathrm{for}(\Sigma_F)$. Then F both preserves and reflects formulae.*

Proof. Since each Σ_F is a signature morphism we can define $\mu_{\Sigma_F} = \mathrm{mod}(\Sigma_F^{\mathrm{op}})$. This ensures the satisfaction condition (1) for each $M' \in \mathrm{mod}(F_{\mathtt{sig}}(\Sigma))$, specifically for those M' compatible with the base axioms $F_{\mathtt{base}}(\Sigma)$, as required for transporting formulae both ways for the STF F. □

Corollary 2. *If an STF reflects formulae, then the $\mu_{\Sigma_F}(M') \models_\Sigma \varphi \Longleftarrow M' \models_{F_{\mathtt{sig}}(\Sigma)} F(\Sigma, \{\varphi\})_2$ part of (2) holds.*

Proof. Since the STF reflects formulae, we know the claimed condition holds for all M' satisfying the base axioms. For those M' that do not hold for the base axioms the claimed condition trivially holds. □

This tells us that when looking for model consistency of an STF we can first check for formulae transportation properties. If the STF transports formulae both ways, we then can explore if the preservation of formulae can be extended to the whole model class.

5.2. Model consistency and formulae transportation for our motivating examples

In this section, we investigate model consistency, reflection and preservation of our STFs within the context of the CASL institution. Fig. 1 summarizes our results. The definition of the STF augment′ can be found below.

It is interesting to see that the union as an STF is not model consistent, i.e., already 'classical structuring operators' fail to work in an institutional setting. Another interesting observation is that all the results summarized in Fig. 1 concern families of patterns. As STFs are parametrized, it is possible to establish properties on them – this is in contrast to the idea of a library of parametrized CASL specifications, cf. our discussion concerning the "Dismissal of the CASL parameterization mechanism" in Section 2.1.1.

Corollary 3. *The STFs* free, generated, lift, *and* union *preserve and reflect properties.*

Proof. By Proposition 9 since signature inclusion is a signature morphism in CASL. □

Neither of these STFs are model consistent since they add non-trivial base axioms. In the special case of lift we remarked, Section 2.4.1, that it also lifts equational formulae from the element to the array level.

Proposition 10. *The* rename *STF is model consistent.*

Proof. Model consistency is inherited from the satisfaction condition of the CASL institution. □

Model consistency also implies preservation and reflection of formulae for the rename STF, cf. Proposition 8.

The error STF modifies the axioms by prepending conditionals. Thus neither of the general insights on model transportation hold. The error STF does not reflect formulae. Consider the ring example from Section 2.5, formula $a * 0 = 0$ for rings, and a model with 3 elements $\{0, 1, \mathrm{error}\}$ as the ring error algebra. Then the formula fails in the ring signature since $\mathrm{error} * 0 = \mathrm{error}$, but for the ring error algebra the translated axiom $a \neq \mathrm{error} \Rightarrow a * 0 = 0$ holds.

Proposition 11. *The* error *STF preserves boolean expression axioms.*[8]

[8] A boolean expression axiom is a quantifier free formulae in first order predicate logic, i.e., with the connectives *and*, *or* and *not*.

Proof. The error STF enlarges the signature, thus there is an embedding from Σ to $\text{error}_{\texttt{sig}}(\Sigma)$ and a related reduct. Let φ be a boolean expression axiom with variables $x_1, \ldots, x_n$ in signature Σ and M' be an algebra in signature $\text{error}_{\texttt{sig}}(\Sigma)$.

Case 1: if φ fails in the reduct of M', then the formula is preserved.

Case 2: Assume the axiom holds in the reduct of M', then the formula $x_1 \neq \text{error} \wedge \ldots \wedge x_n \neq \text{error} \Rightarrow \varphi$ will hold in the error algebra, since the new formula is effectively a formula on a subalgebra of the reduct algebra. □

5.2.1. The STF augment *– or why discrete model categories might be useful*

The last STF that we discuss is the STF augment. Here, it is hard to come up with a suitable 'reduct' functor $\mu_{\Sigma_{\text{augment}}}$, for each signature Σ, since an extra unbounded parameter is introduced as the direction argument. To illustrate this point, we first discuss a new STF augment′.

The STF augment′ is identical to augment, however, the new STF adds one constant, say $a : dir$, to the signature:

$$\begin{aligned}
\text{augment}'_{\langle dir,a,D\rangle,\texttt{sig}}(S,F) &= \langle S \cup \{dir\},\\
&\qquad (F \setminus D) \cup \{a : dir\} \cup \{aug_{\texttt{sig}}(\langle S,F\rangle, dir, d) \mid d \in D \cap F\}\rangle,\\
\text{augment}'_{\langle dir,a,D\rangle,\texttt{base}}(S,F) &= \emptyset,\\
\text{augment}'_{\langle dir,a,D\rangle,\texttt{for},\langle S,F\rangle}(\Phi) &= \{aug_{\texttt{for}}(\langle S,F\rangle, dir, D, \varphi) \mid \varphi \in \Phi\}.
\end{aligned}$$

Proposition 12. *The STF* augment′ *reflects formulae.*

Proof. For each signature Σ a suitable 'reduct' functor $\mu : \text{mod}(\Sigma) \leftarrow \text{mod}(F_{\texttt{sig}}(\Sigma))$ for augment′ needs to select some value for the direction argument. Here we make use of the constant a, and so for an object $M' \in \text{mod}(\text{augment}'_{\langle dir,a,D\rangle,\texttt{sig}}(S,F))$ we define $\mu(M')(s) = M'(s)$ for sort symbols $s \in S$ and

$$\mu(M')(f)(x_1, \ldots x_k) = \begin{cases} M'(f)(x_1, \ldots, x_k);\ f \notin D \\ M'(f)(M'(a), x_1, \ldots, x_k);\ f \in D \end{cases}$$

for function symbols $f : s_1 \times \cdots \times s_k \to s \in F$, $k \geq 0$, $x_i \in \mu(M')(s_i)$, $i = 1 \ldots k$. Given a morphism $h' = (h'_s : M'(s) \to N'(s))_{s \in S \cup \{dir\}} : M' \to N'$ in $\text{mod}(\text{augment}'_{\langle dir,a,D\rangle,\texttt{sig}}(S,F))$, we define its reduct by

$$\mu(h') = (h'_s : M'(s) \to N'(s))_{s \in S} : \mu(M') \to \mu(N').$$

Next we prove that the reduct of a morphism fulfills the homomorphism condition on models (by case distinction).

Case 1: Let $f : s_1 \times \cdots \times s_k \to s \in F \setminus D$ and $x_i \in \mu(M'(s_i)), i = 1 \ldots k$. Then

$$\begin{aligned}
&\mu(h')(\mu(M')(f)(x_1, \ldots, x_k)) = h'_s(M'(f)(x_1, \ldots x_k))\\
=\ &(N'(f))(h'_{s_1}(x_1), \ldots, h'_{s_k}(x_k)) = \mu(N'(f))(h'_{s_1}(x_1), \ldots, h'_{s_k}(x_k)),
\end{aligned}$$

where $h'_{s_i}(x_i) \in N'(s_i) = \mu(N')(s_i)$, $i = 1, \ldots, n$.

Case 2: Let $f : s_1 \times \cdots \times s_n \to s \in D$ and $x_i \in \mu(M'(s_i)), i = 1 \ldots n$. Then

$$\begin{aligned}
&\mu(h')(\mu(M')(f)(x_1, \ldots, x_k)) = h'_s(M'(f)(M'(a), x_1, \ldots x_k))\\
=\ &(N'(f))(h'_{dir}(M'(a)), h'_{s_1}(x_1), \ldots, h'_{s_n}(x_k)) = (N'(f))(N'(a), h'_{s_1}(x_1), \ldots, h'_{s_k}(x_k))\\
=\ &\mu(N'(f))(h'_{s_1}(x_1), \ldots, h'_{s_k}(x_k)),
\end{aligned}$$

where $h'_{s_i}(x_i) \in N'(s_i) = \mu(N')(s_i)$, $i = 1, \ldots, k$.

The STF augment′ with the above choice of μ reflects formulae: Let M' be a model such that $M' \models_{\text{augment}'_{\langle dir,a,D\rangle,sig}(\Sigma)} \text{augment}'_{\langle dir,a,D\rangle,\texttt{base}}(\Sigma)$, thus, when $M' \models aug_{\texttt{for}}(\langle S,F\rangle, dir, D, \varphi)$ we also have $\mu(M') \models \varphi$. □

Proposition 13. *The STF* augment′ *is not formulae preserving.*

Proof. Let $\Sigma = (\{s\}, \{f : s \to s\})$ be a signature. Application of the STF $\text{augment}'_{\text{dir},\{\text{f}\}}$ yields the signature $\Sigma' = (\{s, dir\}, \{a : dir, f : dir \times s \to s\})$. Let $M' \in \text{mod}(\Sigma')$ be the model with

$$M'(s) = \mathbf{N}, \quad M'(dir) = \{1, 2\}, \quad M'(a) = 1, \quad M'(f)(1, x) = x, \quad M'(f)(2, x) = x + 1.$$

Using the same μ as in the previous proof, we have $\mu(M') \models f(x) = x$ while $M' \not\models \forall d : dir \bullet f(d, x) = x$. □

Now we illustrate that it is actually the model morphisms that make it impossible to come up with a 'reduct' μ for the STF augment.

Proposition 14. *The* augment *STF is reflecting formulae when working with discrete model categories.*

Proof. As a 'reduct' functor μ_a define for an object $M' \in \mathrm{mod}(\mathrm{augment}_{\langle dir,D\rangle,\mathtt{sig}}(S,F))$ that $\mu_a(M')(s) = M'(s)$ for sort symbols $s \in S$ and

$$\mu_a(M')(f)(x_1,\ldots x_n) = \begin{cases} M'(f)(x_1,\ldots,x_n);\ f \notin D \\ M'(f)(a,x_1,\ldots,x_n);\ f \in D \end{cases}$$

for function systems $f : s_1 \times \cdots \times s_n \to s \in F$, $n \geq 0$, $x_i \in \mu_a(M')(s_i)$, $i = 1 \ldots n$ and an arbitrary value $a \in M'(dir)$.

Proof by case distinction.

- There exists no model M' with $M' \models \mathrm{augment}_{\langle dir,D\rangle,\mathtt{sig}}(\varphi)_2$. Then the premise is false and the implication holds.
- There exists a model M' with $M' \models \mathrm{augment}_{\langle dir,D\rangle,\mathtt{sig}}(\varphi)_2$. In case that φ involves a function symbol $f \in D$, $\mathrm{augment}_{\langle dir,D\rangle,\mathtt{sig}}(\varphi)$ is of the form $\forall d : dir \bullet \varphi'$, i.e., the formula holds for any direction. Thus, it holds for any specific choice, and in particular for a. □

The STF augment on the discrete model category fails to preserve formulae for the same reason augment′ fails to preserve formulae.

The choice of arbitrary values used to define the discrete reduct μ above is completely uncoordinated between models and between homomorphisms on the models. This causes problems when we try to map the homomorphisms. Consider a signature $\Sigma = (\{s\}, \{f : s \to s\})$. Application of the STF $\mathrm{augment}_{\mathrm{dir},\{\mathrm{f}\}}$ yields the signature $\Sigma' = (\{s, dir\}, \{f : dir \times s \to s\})$. Let $M' \in \mathrm{mod}(\Sigma')$ be the model with

$$M'(s) = \{*, +\}, \quad M'(f)(1,x) = *, \quad M'(f)(2,x) = +, \quad M'(dir) = \{1,2\},$$

let $h', k' : M' \to M'$ be two homomorphisms, identical everywhere, e.g., $h'_s = k'_s = id_s$ for $s \in S \setminus \{dir\}$, except for the mapping of dir where

$$h'_{dir}(1) = 1, \quad h'_{dir}(2) = 2, \quad k'_{dir}(1) = 2, \quad k'_{dir}(2) = 1.$$

If we now choose $\mu(M')(f)(x) = M'(f)(1,x)$, we can establish the homomorphism condition for the reduct of h', as $h'_{dir}(1) = 1$:

$$\begin{aligned} & \mu(h')(\mu(M')(f)(x)) = h'_s((M'(f)(1,x)) \\ = \ & (M'(f))(h'_{dir}(1), h'_{s_1}(x)) = M'(f)(1, h'_s(x)) \\ = \ & \mu(M'(f))(h'_s(x)). \end{aligned}$$

However, we fail with the homomorphism condition for the reduct of k', as $k'_{dir}(1) = 2$:

$$\begin{aligned} & \mu(k')(\mu(M')(f)(x)) = k'_s((M'(f)(1,x)) \\ = \ & (M'(f))(k'_{dir}(1), h'_{s_1}(x)) \neq M'(f)(1, k'_s(x)) \\ = \ & \mu(M'(f))(k'_s(x)). \end{aligned}$$

The model category for the STF augment′ enforces the equation $h'_{dir}(a) = M'(a)$ for all model morphisms h'. Having no constant a around gives more freedom to the model morphisms – and we run into trouble defining their reducts.

In case we would work with a discrete model category, there is no problem of choosing values for the direction which are 'consistent' with model homomorphisms – for the simple reason that there are no nontrivial model homomorphisms.

5.3. Institution construction from model consistent STFs

Recall the construction of a path category. If we start with a collection of objects and arcs (morphisms without a composition rule), we can create a *path category* by reusing the objects, and defining every path of such arcs (including the empty path and the singleton path) as a morphism. Path composition is concatenation. It has all the necessary properties: the empty path at each object being the identity morphism and associativity inherited from concatenation. If the collection of arcs is empty, the path constructions yields the discrete category.

Utilizing the idea of a path category, we form an *STF path institution* $\mathcal{INST}_{\mathcal{F}}$ relatively to a collection $\mathcal{F}$ of model-consistent STFs, with reduct functors μ_{Σ_F} for each signature Σ and each STF $F \in \mathcal{F}$, over a given institution $\mathcal{INST}$:

- Its category of signatures is the path category $\mathbf{Sig}_{\mathcal{F}}$, where we take the signatures Σ as objects and, for every $F \in \mathcal{F}$, the non-trivial $\Sigma_F : \Sigma \to F_{\mathtt{sig}}(\Sigma)$ as the arcs.
- Its functor $\mathrm{for}_{\mathcal{F}} : \mathbf{Sig}_{\mathcal{F}} \to \mathbf{Set}$ of formulae is given on objects by $\mathrm{for}_{\mathcal{F}}(\Sigma) = \mathrm{for}(\Sigma)$. For $F \in \mathcal{F}$ and any non-trivial Σ_F, we define $\mathrm{for}_{\mathcal{F}}(\Sigma_F)(\Phi) = F(\Sigma, \Phi)_2$. This extends to paths in $\mathbf{Sig}_{\mathcal{F}}$ by function composition in **Set**.

- Its contravariant functor $\mathrm{mod}_{\mathcal{F}} : \mathbf{Sig}_{\mathcal{F}}^{\mathrm{op}} \to \mathbf{CAT}$ of models is given on signatures by $\mathrm{mod}_{\mathcal{F}}(\Sigma) = \mathrm{mod}(\Sigma)$ from the underlying institution. As every $F \in \mathcal{F}$ is model consistent, we can define $\mathrm{mod}_{\mathcal{F}}(\Sigma_F^{\mathrm{op}} : \Sigma \leftarrow F_{\mathtt{sig}}(\Sigma)) = \mu_{\Sigma_F}$, extended to paths in $\mathbf{Sig}_{\mathcal{F}}$ by functor composition in **CAT**.
- Its satisfaction relation is the one from $\mathcal{INST}$.

Theorem 4 *(STF path institution). A collection $\mathcal{F}$ of model consistent STFs on the discrete theory category associated to an institution $\mathcal{INST}$ induces an* STF path institution $\mathcal{INST}_{\mathcal{F}} = \langle \mathbf{Sig}_{\mathcal{F}}, \mathrm{for}_{\mathcal{F}} : \mathbf{Sig}_{\mathcal{F}} \to \mathbf{Set}, \mathrm{mod}_{\mathcal{F}} : \mathbf{Sig}_{\mathcal{F}}^{\mathrm{op}} \to \mathbf{CAT}, \models \rangle$.

Proof. The constructions above for model consistent STFs ensures the functors have the appropriate sources and targets, and have the needed properties with respect to $\models$ for $\mathcal{INST}_{\mathcal{F}}$ to be an institution. □

Some collections of STFs $\mathcal{F}$ may have morphism composition properties that are not reflected in the STF path signature category $\mathbf{Sig}_{\mathcal{F}}$. For instance, consider a signature morphism which has an inverse. Then the two renaming STFs corresponding to the signature morphism and its inverse will compose to the identity functor $\mathrm{id}_{\mathbf{Th}_{\mathrm{id}}}$ on $\mathbf{Th}_{\mathrm{id}}$, but as paths the composition of the two renamings will not be $\mathrm{id}_{\mathbf{Th}_{\mathrm{id}}}$. Since each path is a sequence of non-trivial arcs indexed by the corresponding STFs, we can look at the composition of the underlying STFs in order to identify paths which behave the same on the corresponding theories. The empty path, the identity, on a signature corresponds to the identity STF.

We define an equivalence relation on paths in the STF path institution. Let $p, q : \Sigma \to \Sigma'$ be paths with associated STFs $F_1, \ldots F_n, n \geq 0$, and $G_1, \ldots G_m, m \geq 0$, resp. The paths p and q are equivalent w.r.t. $\mathcal{F}$ iff for all theories $\langle \Sigma, \Phi \rangle$ the resulting theories $(F_1; \ldots; F_n)(\Sigma, \Phi) = (G_1; \ldots; G_m)(\Sigma, \Phi)$.

Using this notion, we define the category $\mathbf{Sig}_{/\mathcal{F}}$ as the quotient category of $\mathbf{Sig}_{\mathcal{F}}$, i.e., they both have the same objects (the signatures), but each morphism $p_{/\mathcal{F}} : \Sigma \to \Sigma'$ in $\mathbf{Sig}_{/\mathcal{F}}$ is the collection of all paths $p : \Sigma \to \Sigma'$ in $\mathbf{Sig}_{\mathcal{F}}$ where the underlying composition of the STFs are equal for all theories. Note that the composition of STFs from $\mathcal{F}$ does not have to yield STFs in $\mathcal{F}$.

The formula functor $\mathrm{for}_{/\mathcal{F}} : \mathbf{Sig}_{/\mathcal{F}} \to \mathbf{Set}$ on quotient paths $\mathbf{Sig}_{/\mathcal{F}}$ follows from $\mathrm{for}_{\mathcal{F}} : \mathbf{Sig}_{\mathcal{F}} \to \mathbf{Set}$. They are identical on objects (signatures) and for any pair of paths p, q in a quotient path we have that $\mathrm{for}_{\mathcal{F}}(p)(\Phi) = \mathrm{for}_{\mathcal{F}}(q)(\Phi)$ by construction of the quotient path.

When all STFs in $\mathcal{F}$ are model consistent, then $\mathrm{mod}_{/\mathcal{F}} : \mathbf{Sig}_{/\mathcal{F}}^{\mathrm{op}} \to \mathbf{CAT}$ is similarly defined from $\mathrm{mod}_{\mathcal{F}} : \mathbf{Sig}_{\mathcal{F}}^{\mathrm{op}} \to \mathbf{CAT}$. Again the construction of the quotient paths in $\mathbf{Sig}_{/\mathcal{F}}$ ensures well definedness.

Theorem 5 *(STF institution). A collection $\mathcal{F}$ of model consistent STFs on the discrete theory category associated to an institution $\mathcal{INST}$ induces an* STF institution $\mathcal{INST}_{/\mathcal{F}} = \langle \mathbf{Sig}_{/\mathcal{F}}, \mathrm{for}_{/\mathcal{F}} : \mathbf{Sig}_{/\mathcal{F}} \to \mathbf{Set}, \mathrm{mod}_{/\mathcal{F}} : \mathbf{Sig}_{/\mathcal{F}}^{\mathrm{op}} \to \mathbf{CAT}, \models \rangle$.

Proof. The constructions above for model consistent STFs ensures the functors have the appropriate sources and targets. The needed properties with respect to $\models$ follows from the STF path institution. □

An STF institution is more 'compact' than the STF path institution since multiple representations of the same signature morphisms have been eliminated.

Both these institutions are additive with respect to the selection of STFs. Let $\mathcal{INST}$ be an institution with $\mathcal{F}_1$ and $\mathcal{F}_2$ being collections of model consistent STFs on the associated theory category. Then $\mathcal{INST}_{/\mathcal{F}_1}$, $\mathcal{INST}_{/\mathcal{F}_2}$ and $\mathcal{INST}_{/\mathcal{F}_1 \cup \mathcal{F}_2}$ are STF institutions.

This allows us to build up an institution $\mathcal{INST}_{/\mathcal{F}}$ step by step, with powerful signature morphisms created from the non-trivial arcs of $\mathcal{F}$. The starting point can be an institution with a discrete signature category, extended with more versatile signature morphisms as they are discovered. Each such addition is by construction connected to a syntactic specification structuring mechanism which can be freely composed with other syntactic specification structuring mechanisms.

Such flexibility is useful when a domain expert starts organising a domain, e.g., as an ontology. An ontology expert may come in at a later stage, suggesting standard or novel syntactic theory functors in collaboration with the domain expert, providing structure (as a series of institutions) and improved reuse capabilities (as collections of STFs) in a consistent and extensible STF framework.

6. Pragmatics of developing OMSs

Reflecting on the framework we have developed above, we can make the following observations on developing ontologies, models and formal specifications (OMSs). For a specification, modelling or ontology formalism, the institution notion captures its essential aspects:

- The (discrete) signature category defines the notation of the formalism.
- The formulae functor defines the formulae that can be expressed for each signature.
- The model functor defines the possible semantics of a signature.
- The satisfaction relation tells when a semantical model for a signature is compatible with a specific formulae.

The requirements on the institution ensures that whenever there is a mapping between notations of the specification formalism, then the truth of the formulae remains invariant to the change of notation. Coming up with good notions of signature morphisms (notation mappings) that have the required properties is hard due to these requirements.

The STF notion formalises parts of the process of developing large specifications from smaller components. STFs map signatures to signatures, possibly adding base axioms, and allow modifications to formulae by systematically changing them to match the new notation. Some of these STFs are truth invariant (model consistent), and thus provide an apparatus similar to signature morphisms in an institution, others deliberately change the model classes in model inconsistent ways. The latter, like adding error elements to a specification, are not truth-invariant development steps.

Thus specification development along a signature morphism ensures reuse of truth, while other development steps need to be investigated on a case by case basis to identify preserved, reflected or in other ways induced properties. With STFs these insights can be gathered step by step, giving the specification or ontology developer a more powerful toolset as the development process becomes better understood.

Consider the CASL institution [15]. It is a powerful institution in the sense that the CASL rename facility goes far beyond standard practice in most algebraic specification languages. As such the research leading up to this institution is significant. Using the methodology developed in this paper, we could grow this institution piecemeal instead.

The discrete CASL institution would follow from defining the CASL specification language and its semantics. Then the introduction of STFs that allow renaming of type names, similar to simple uses of templates in C++ and generics in Java, can easily be shown to be model consistent. This then gives a CASL institution where truth is invariant of type names. Independently of this, STFs that allow changes to function names can be investigated and shown to be model consistent, giving a different institution for CASL specification reuse. Further, STFs that allow swapping of function argument ordering can be shown to be model consistent, giving a CASL institution where the reordering of function arguments is the signature morphism notion. These STFs can then be combined, giving the standard CASL institution [15] with signature morphisms that allow changing of type and function names as well as function argument reordering.

We can then conceive of going beyond this CASL institution by investigating STFs corresponding to "wrapper functions", commonly needed in programming languages like C++ in order to adapt APIs. Wrapper functions would expand to a new kind of signature morphism which would allow changing the number of arguments to functions. These can also be shown to be model consistent, hence giving a CASL institution with an expressive power far beyond the standard CASL institution. Again, adding this signature morphism to the repertoire of CASL will be compatible with all existing CASL signature morphisms per Theorem 5.

A similar sequence of exploration steps will be valid for other OMS domains, giving a pragmatic approach to growing the underlying institution, with its benefits of truth being invariant of notation, as well as building a useful toolset of STFs for compact specifications in any such formalism.

7. Related work

There have been a number of different approaches which have tried to address the challenge posed in this paper: how to characterize a class of transformations of specifications which are

- expressive enough to be useful in practice,
- tame enough to be compatible with semantical frameworks.

Forssell et al. utilize substitution in order to define the effect of actual parameters on formal parameters of templates [18]. As we are also interested in enlarging signatures, cf. the example on lifting as presented in Section 2.4, replacing existing symbols is not general enough.

A number of solutions have been proposed in a language-independent way using institution theory. Castro et al. provide a general specification framework for dynamically reconfigurable systems [19]. In this context, they study how to transform the specification of a component into the specification of a component manager. To this end, they utilize the concept of so-called representation maps [20] as these allow to change the arities of symbols. Similarly, in the context of the specification language Z, the concept of promotion – that can be used to conveniently define large specifications from collections of simpler ones – can be captured as an institution representation [21]. The difference to our work is that representation maps preserve the properties of the original specification, while we are also interested in a framework that allows also to destroy properties, see the examples presented in Section 2.4 and accompanying STFs in Section 3.4.

Diaconescu [22] introduces a notion of a structured institution ℓ' over an institution ℓ through a functor Φ from the signature category of ℓ' to the signature category of ℓ with some additional constraints. Most of the structuring mechanisms relate diagrams in ℓ' to diagrams in ℓ through Φ. A classical example of a structured institution is when ℓ' is an *institution of ℓ-theories*. The syntactic theory functors allow us to create a similar theory-based institution ℓ' for any ℓ. However, the STF constructed ℓ' will in general have a discrete signature category, thus it is without any interesting diagrams. Diaconescu's results on normal forms and lifting of entailment systems (proof systems) from ℓ to ℓ' are still valid in the STF context, and can be used to prove properties directly on STF specifications by exploiting the tools for the underlying institution.

See also the discussion of the STFs' relationship to theoroidal and generalised theoroidal maps in Section 3.2.

8. Summary and future work

In this paper we introduce the notion of a syntactic theory functor (STF) as an institution-independent construction. Institution-independence means STFs are applicable to ontologies, modelling and formal specifications (OMS). STFs add reuse capabilities to an institution, capabilities that can be flattened to basic specifications in the institution. An STF can therefore be understood in terms of the basic specification patterns it supports.

The paper explores this new structuring device both at a formal, institution-independent level, and with concrete examples in the algebraic specification formalism CASL.

At the general, institution-independent level we show that STFs are signature preserving and sensible, and that they in general compose and interoperate in natural ways. STFs subsume the standard set of institution-independent reuse mechanisms. We also provide insights on how formulae of the component specifications are preserved or reflected by the specification constructed using an STF. We further prove that model consistent STFs can be used to grow an institution: building a more advanced institutions stepwise from a simple, even discrete, one. This can be very important when developing, e.g., a new ontology. Then exploring reuse in the form of STFs for the ontology can give the ontology formalism a richer structure and better interoperability with other institutions in the setting of DOL [7]. This can enable using tools originally built for a different institution in the new one.

At the pragmatical level we have given many examples of STFs solving practical specification problems in algebraic specifications. This includes declaring specifications with many distinct constants, lifting specification of functions from elements to arrays, adding extra arguments to functions in specifications, and showing that STFs can be used to deal with partial algebras in a total setting, e.g., by systematically introducing an error element. As discussed in [10] such encodings are error prone and thus should be supported by a tool like STFs to avoid repeating the pitfalls.

In future work we plan to establish a library of concrete STFs together with their properties, e.g., what aspects of a specification they preserve, reflect or if they are model preserving. Especially interesting are STFs for encoding partiality in a total setting. Further, we want to develop a methodology and possibly tool support for specifying with STFs. While our endeavours will focus on algebraic specifications, e.g., CASL, we would hope that – taking, e.g., the interest in design pattern in the ontology community into account – our idea of STFs will be taken up in various other specification contexts.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

We would like to thank Erwin R. Catesbeiana (Jr.) for sending us on the route of path categories. Furthermore, we want to thank the reviewers for their thorough reviews and helpful suggestions of how to improve our paper.

References

[1] E. Gamma, R. Helm, R. Johnson, V. John, Design Patterns, Addison-Wesley, 1994.
[2] P.D. Mosses (Ed.), CASL Reference Manual: The Complete Documentation of the Common Algebraic Specification Language, Lecture Notes in Computer Science, vol. 2960, Springer, 2004.
[3] M. Cerioli, G. Reggio, Basic CASL at work: a compass for the labyrinth of partiality, subsorting and predicates, presentation at WADT, 1999.
[4] M. Roggenbach, T. Mossakowski, Methodological guidelines, coFI Note M-6, Version 0.7, 2002.
[5] C. Alexander, S. Ishikawa, M. Silverstein, A Pattern Language, Oxford University Press, 1977.
[6] B. Krieg-Brückner, T. Mossakowski, F. Neuhaus, Generic ontology design patterns at work, CoRR, arXiv:1906.08724, 2019, http://arxiv.org/abs/1906.08724.
[7] T. Mossakowski, M. Codescu, F. Neuhaus, O. Kutz, The distributed ontology, modeling and specification language – dol, in: A. Koslow, A. Buchsbaum (Eds.), The Road to Universal Logic, Vol. 2, Birkhäuser, 2015, pp. 489–520.
[8] T. Mossakowski, C. Maeder, K. Lüttich, The heterogeneous tool set, hets, in: O. Grumberg, M. Huth (Eds.), Tools and Algorithms for the Construction and Analysis of Systems, 13th International Conference, TACAS 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007, Braga, Portugal, March 24 - April 1, 2007, Proceedings, in: Lecture Notes in Computer Science, vol. 4424, Springer, 2007, pp. 519–522.
[9] M. Haveraaen, H.A. Friis, H. Munthe-Kaas, Computable scalar fields: a basis for PDE software, J. Log. Algebraic Methods Program. 65 (1) (2005) 36–49, https://doi.org/10.1016/j.jlap.2004.12.001.
[10] P.D. Mosses, The use of sorts in algebraic specifications, in: M. Bidoit, C. Choppy (Eds.), Recent Trends in Data Type Specification, 8th Workshop on Specification of Abstract Data Types Joint with the 3rd COMPASS Workshop, Dourdan, France, August 26–30, 1991, Selected Papers, in: Lecture Notes in Computer Science, vol. 655, Springer, 1991, pp. 66–92.
[11] J.L. Fiadeiro, Categories for Software Engineering, Springer, 2005.
[12] J.A. Goguen, G. Rosu, Institution morphisms, Form. Asp. Comput. 13 (3–5) (2002) 274–307, https://doi.org/10.1007/s001650200013.
[13] M. Codescu, Generalized theoroidal institution comorphisms, in: A. Corradini, U. Montanari (Eds.), Recent Trends in Algebraic Development Techniques, 19th International Workshop, WADT 2008, Pisa, Italy, June 13–16, 2008, in: Lecture Notes in Computer Science, vol. 5486, Springer, 2008, pp. 88–101.
[14] D. Sannella, A. Tarlecki, Specifications in an arbitrary institution, Inf. Comput. 76 (2) (1988) 165–210, https://doi.org/10.1016/0890-5401(88)90008-9.
[15] T. Mossakowski, Relating CASL with other specification languages: the institution level, Theor. Comput. Sci. 286 (2) (2002) 367–475, https://doi.org/10.1016/S0304-3975(01)00369-3.
[16] F. Orejas, Structuring and modularity, in: E. Astesiano, H. Kreowski, B. Krieg-Brückner (Eds.), Algebraic Foundations of Systems Specification, IFIP State-of-the-Art Reports, Springer, 1999, pp. 159–200.

[17] J. Bergstra, J. Tucker, Algebraic specifications of computable and semicomputable data types, Theor. Comput. Sci. 50 (2) (1987) 137–181, https://doi.org/10.1016/0304-3975(87)90123-X.
[18] H. Forssell, D.P. Lupp, M.G. Skjaeveland, E. Thorstensen, Reasonable macros for ontology construction and maintenance, poster paper presented at the Description Logics workshop, 2017.
[19] P.F. Castro, N.M. Aguirre, C.G. López Pombo, T.S.E. Maibaum, Towards managing dynamic reconfiguration of software systems in a categorical setting, in: A. Cavalcanti, D. Deharbe, M.-C. Gaudel, J. Woodcock (Eds.), Theoretical Aspects of Computing – ICTAC 2010, Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, pp. 306–321.
[20] A. Tarlecki, Moving between logical systems, in: M. Haveraaen, O. Owe, O. Dahl (Eds.), Recent Trends in Data Type Specification, 11th Workshop on Specification of Abstract Data Types Joint with the 8th COMPASS Workshop, Oslo, Norway, September 19–23, 1995, Selected Papers, in: Lecture Notes in Computer Science, vol. 1130, Springer, 1995, pp. 478–502.
[21] P.F. Castro, N. Aguirre, C.L. Pombo, T.S.E. Maibaum, Categorical foundations for structured specifications in Z, Form. Asp. Comput. 27 (5) (2015) 831–865, https://doi.org/10.1007/s00165-015-0336-0.
[22] R. Diaconescu, An axiomatic approach to structuring specifications, Theor. Comput. Sci. 433 (2012) 20–42, https://doi.org/10.1016/j.tcs.2012.03.001.